

framework training
We love technology

Introduction to Commercial Software Development

Informed Academy

020 3137 3920

@FrameworkTrain

frameworktraining.co.uk

Welcome



Toby Dussek

Informed Academy



framework training
business value through education

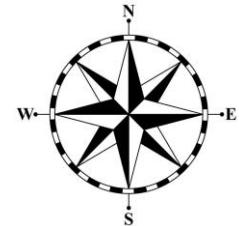
Introductions

- Course Instructor

- Toby Dussek

- Course attendees

- You...
 - Name
 - Professional Position
 - Java, JavaScript & Programming experience
 - Experience with Git and Maven
 - Database experience
 - Motivation and Expectations



Introduction

□ Course objectives

- Developer Landscape
- Use of Git
- Java Technologies
 - classes, objects, constructors, methods, inheritance, abstraction, interfaces, functions, streams, collections, exception handling
- Dependency Management with Maven
- Class Design
- Testing and JUnit

Questions?



Developer Landscape

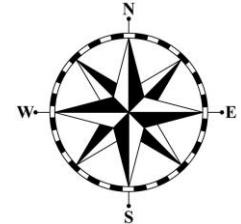


Toby Dussek

Informed Academy



framework training
business value through education



Plan for Session

- Explore the Developer Landscape
 - Operating systems
 - Programming Languages
 - Editors
 - Library Management
 - Code Source code and Version Control
 - Building Software
 - Testing Environments
 - Task Management
 - Developer Documentation

Developer Landscape

- You and your machine



But what operating systems does it run?

Developer Landscape

□ You and your machine



Windows



Windows 10



MacOS

MacOS

Linux



But what operating systems does it run?

Developer Landscape

□ You and your machine



Via Desktop GUI



Via Command Line

```
$ mkdir my_project
$ cd my_project
$ git init
Initialized empty Git repository in my_project/.git/
$
```

How do you interact with it?

Programming Languages

- Developing Code



What Programming Languages are used?

Programming Languages

- Developing Code



What Programming Languages are used?

Programming Languages

□ Developing Code



TypeScript



JavaScript



Python



What Programming Languages are used?

Programming Languages

□ Developing Code



JavaScript



TypeScript



Python



Kotlin



What Programming Languages are used?

Programming Languages

□ Developing Code



TypeScript



JavaScript



Python



Kotlin



Scala

What Programming Languages are used?

Programming Languages

□ Developing Code



TypeScript



JavaScript



Python



Kotlin



Scala

What Programming Languages are used?

Developer Landscape

- You and your machine



What Editor / IDE do you use?

Developer Landscape

- You and your machine



Emacs



Vim



What Editor / IDE do you use?

Developer Landscape

- You and your machine



PyCharm



Emacs



Vim



WebStorm



What Editor / IDE do you use?

Developer Landscape

□ You and your machine



PyCharm



Emacs



Vim



WebStorm



Eclipse



NetBeans



IntelliJ



What Editor / IDE do you use?

Developer Landscape

- You and your machine



Visual Studio



Visual Studio Code



PyCharm



WebStorm



NetBeans



Emacs



Vim



Eclipse



What Editor / IDE do you use?

An IDE



Integrated
Development
Environment

The screenshot shows a dark-themed IDE interface. On the left is an Explorer sidebar with a tree view of files and folders. The main area is a code editor titled "query-script.js" with the following content:

```
query-script.js — javascript-intro
12-ajax > js > JS query-script.js > ⚡ ready() callback
1   $(document).ready(function() {
2     console.log('Setting up query form');
3     $("#query-form").submit(function(event) {
4       console.log('Running submit');
5       $.post("http://localhost:3000", $(this).serialize(), function(play) {
6         console.log('In post callback function - ', play);
7         var html = "<div class='play'>";
8         html += "<h3 class='title'>" + play.name + "</h3>";
9         html += "Written: " + play.date;
10        html += "<br>Category: " + play.category;
11        html += "<br>Synopsis: " + play.synopsis;
12        html += "</div>";
13        $("#plays").append($(html));
14      });
15      event.preventDefault();
16    });
17  });
18
```

The status bar at the bottom shows: master, 0, 0 △ 0, Ln 2, Col 38, Spaces: 2, UTF-8, LF, JavaScript, Prettier, and a few small icons.

Developer Landscape

- You and your machine



Library Dependencies

What about libraries?

Developer Landscape

- You and your machine



Library Dependencies

Maven  **Maven™**

npm



Gradle



What about libraries?

Developer Landscape

- You and your machine



locally

Where do you store your data?

Developer Landscape

- You and your machine



locally



Files

Relational
Database



NoSQL Database



Where do you store your data?



remotely
(database server)

Developer Landscape

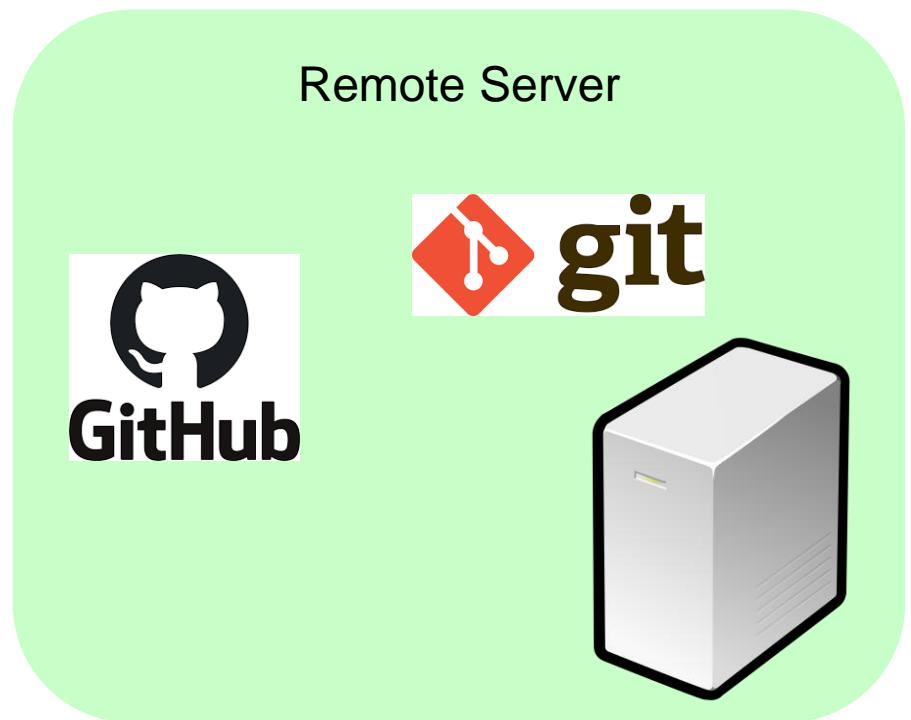
□ You and your machine



*commit
changes*



Local Client



Where do you store your code?

Public GitHub Web Interface

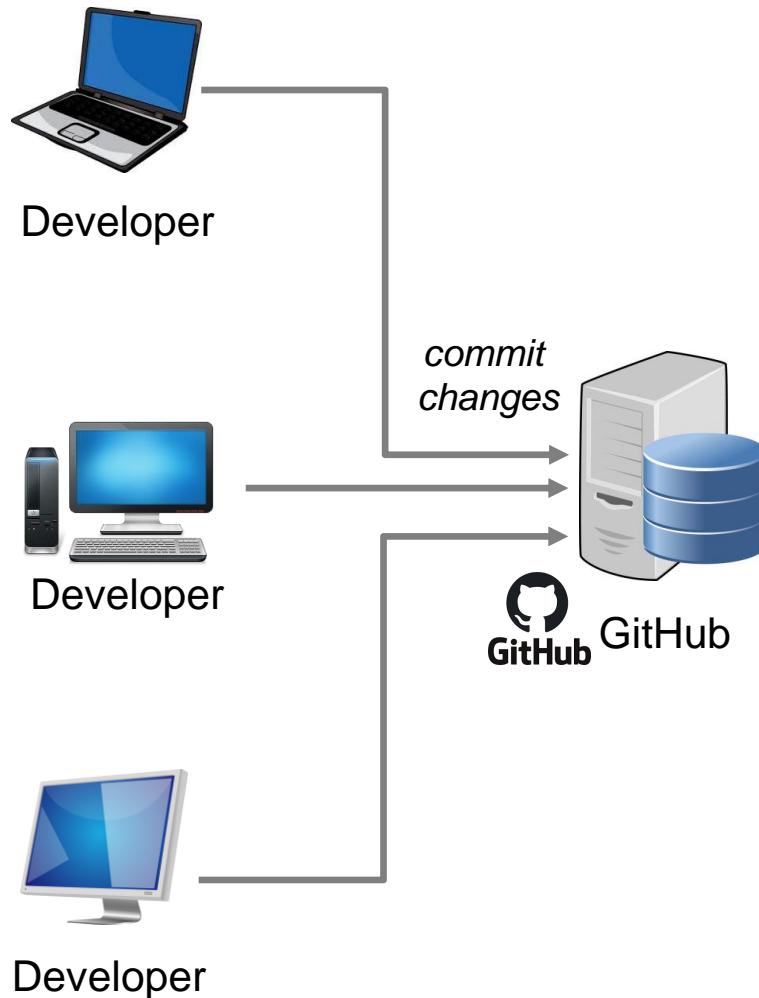
The screenshot shows a GitHub repository page for 'johnehunt/javascript-intro'. The page includes the repository name, a brief description ('An introductory course for JavaScript'), and various metrics like 56 commits, 1 branch, 0 packages, 0 releases, 1 contributor, and Apache-2.0 license. A list of recent commits is displayed, each with a commit message, author, and timestamp.

Commit	Message	Time Ago
John Hunt Added a couple of simple test examples	Latest commit ebb5269	3 hours ago
01-helloworld	Moved html page examples	3 days ago
02-js-variables	Moved comparison operators	yesterday
03-flow-of-control	Layout changes, fixed typos in comments and re organized order of exa...	yesterday
04-functions	Modified some comments	yesterday
05-objects	Added example of a class (static) property	23 hours ago
06-inheritance	Tidied toString() and fixed a couple of comments	23 hours ago
07-arrays	Reorded examples	22 hours ago
08-errors	Added error handling examples	3 days ago
09-js-in-web-pages	tidied script tags	20 hours ago
10-html-dom	Simplified example	6 hours ago
11-jquery	Added a title to the page	6 hours ago
12-ajax	layout changes	5 hours ago
12-server	Example of using ajax style requests in jQuery	3 days ago

How do you Build Systems?

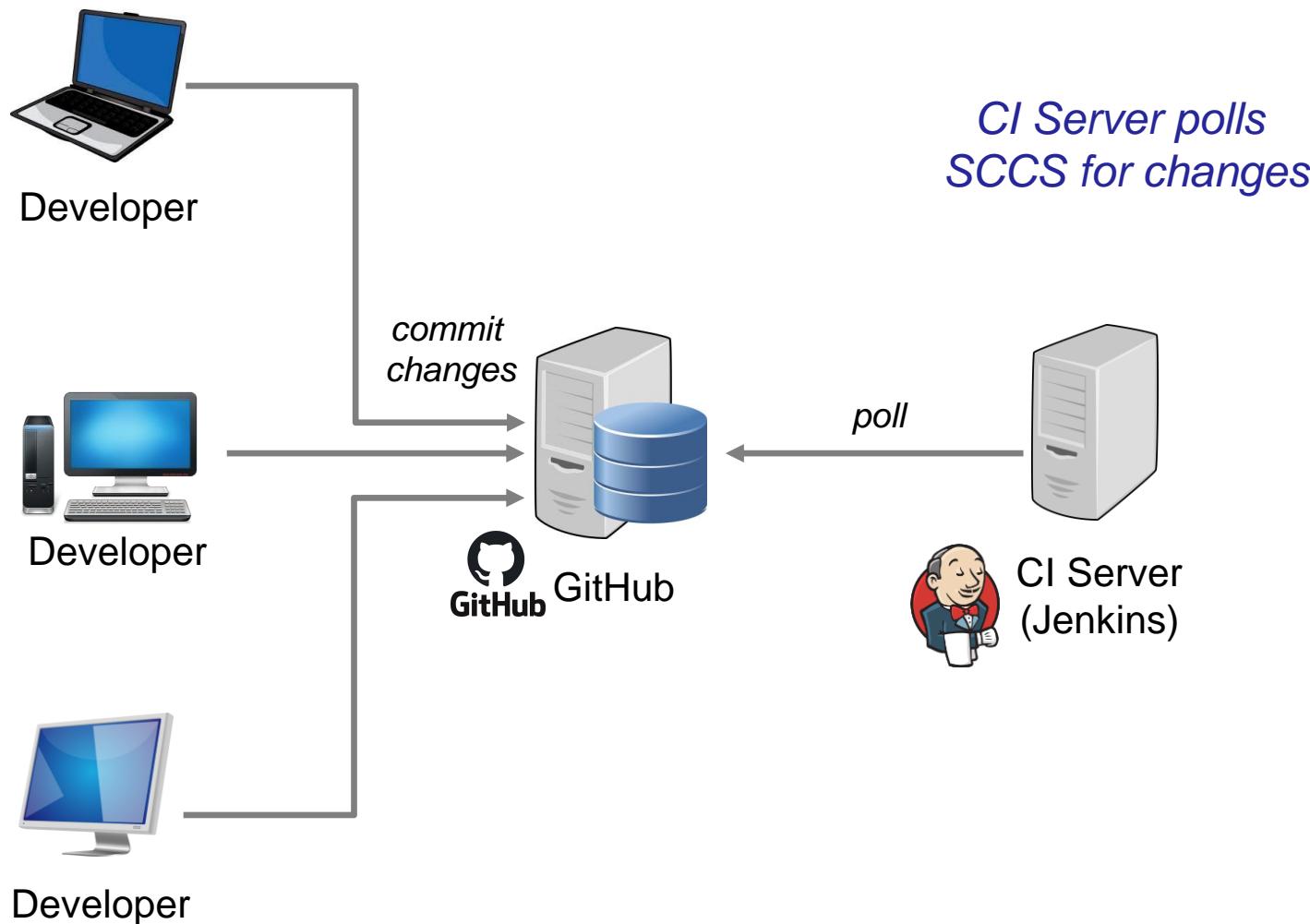


Continuous Integration Infrastructure

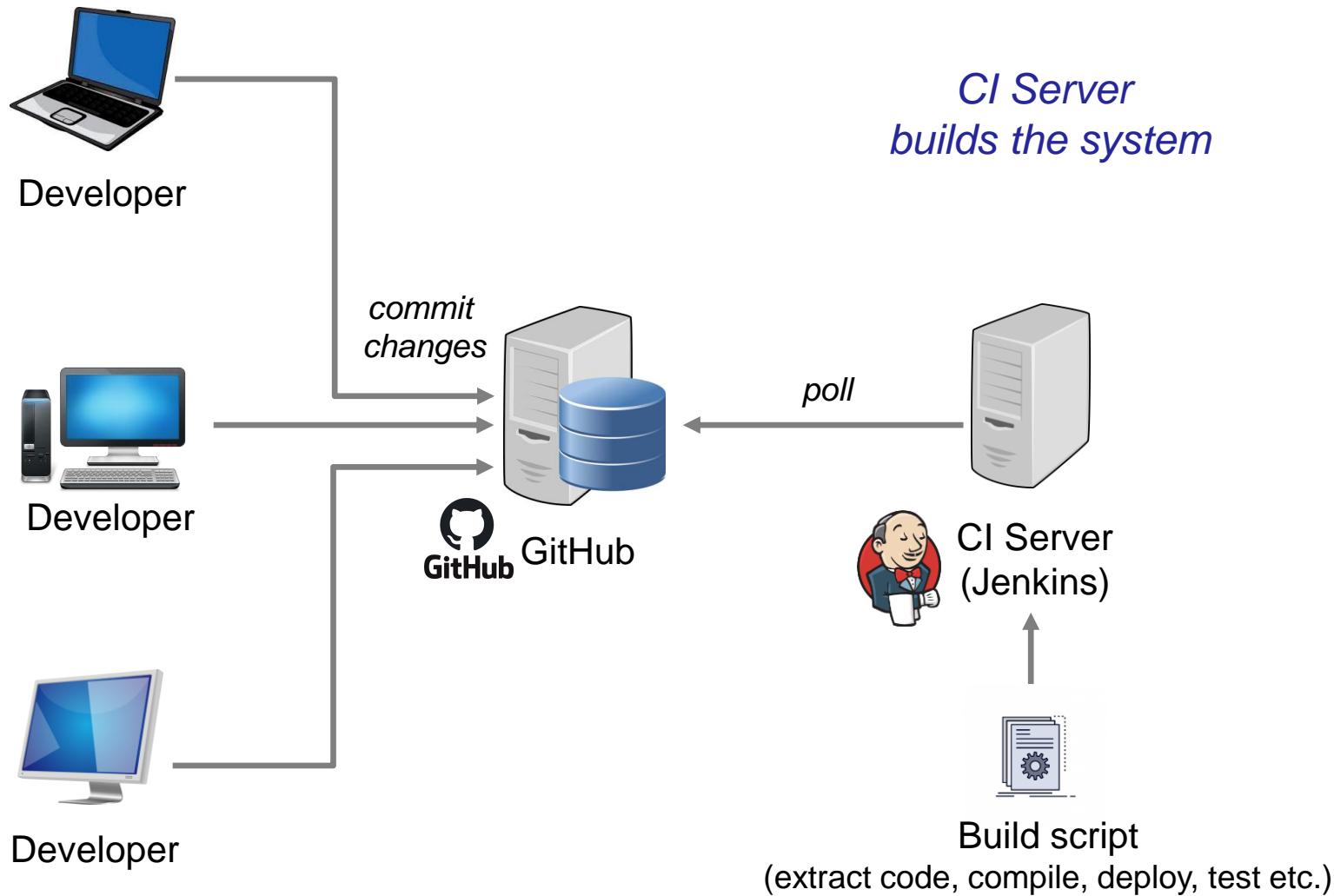


*All code is held
in SCCS*

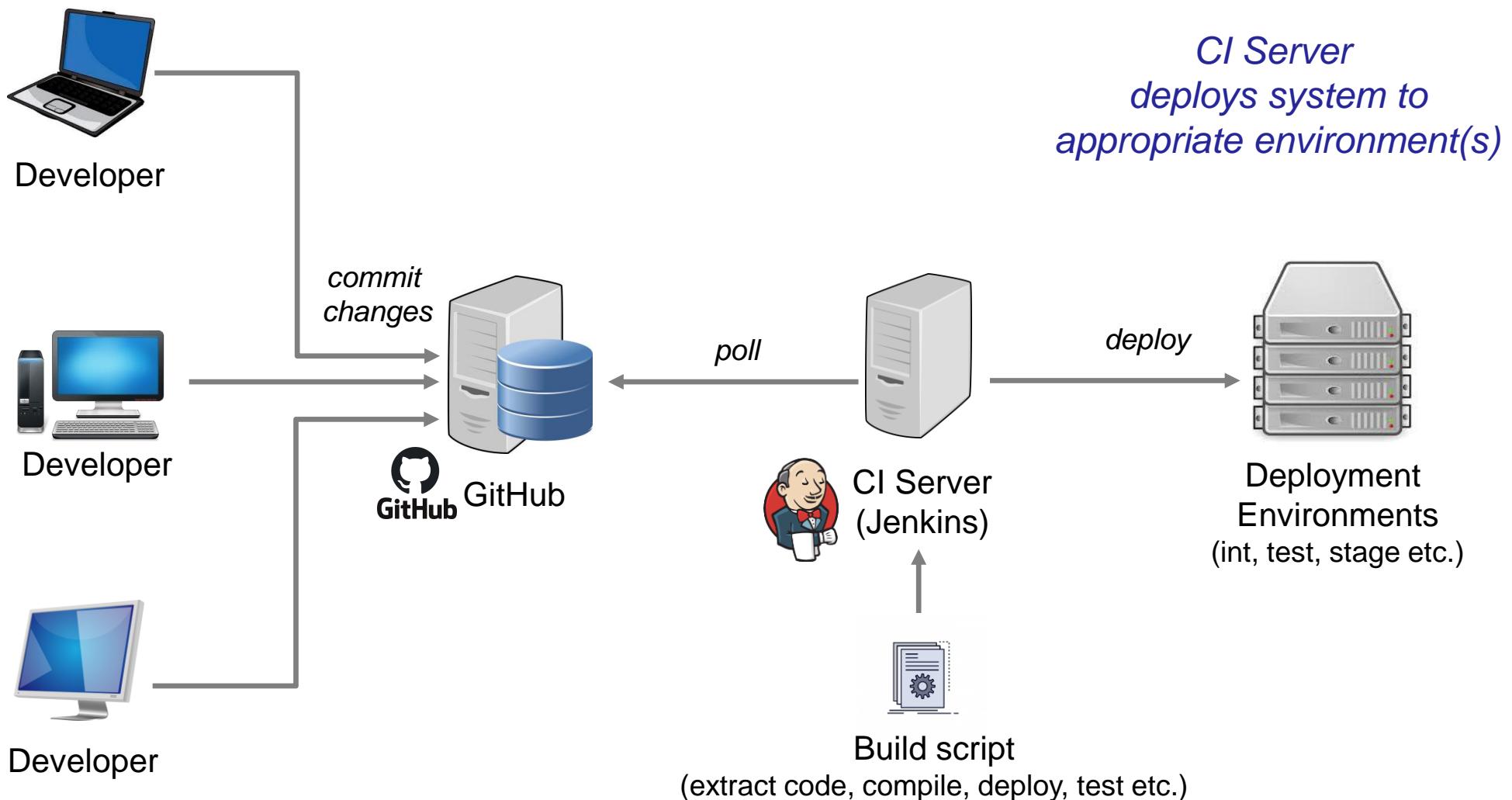
Continuous Integration Infrastructure

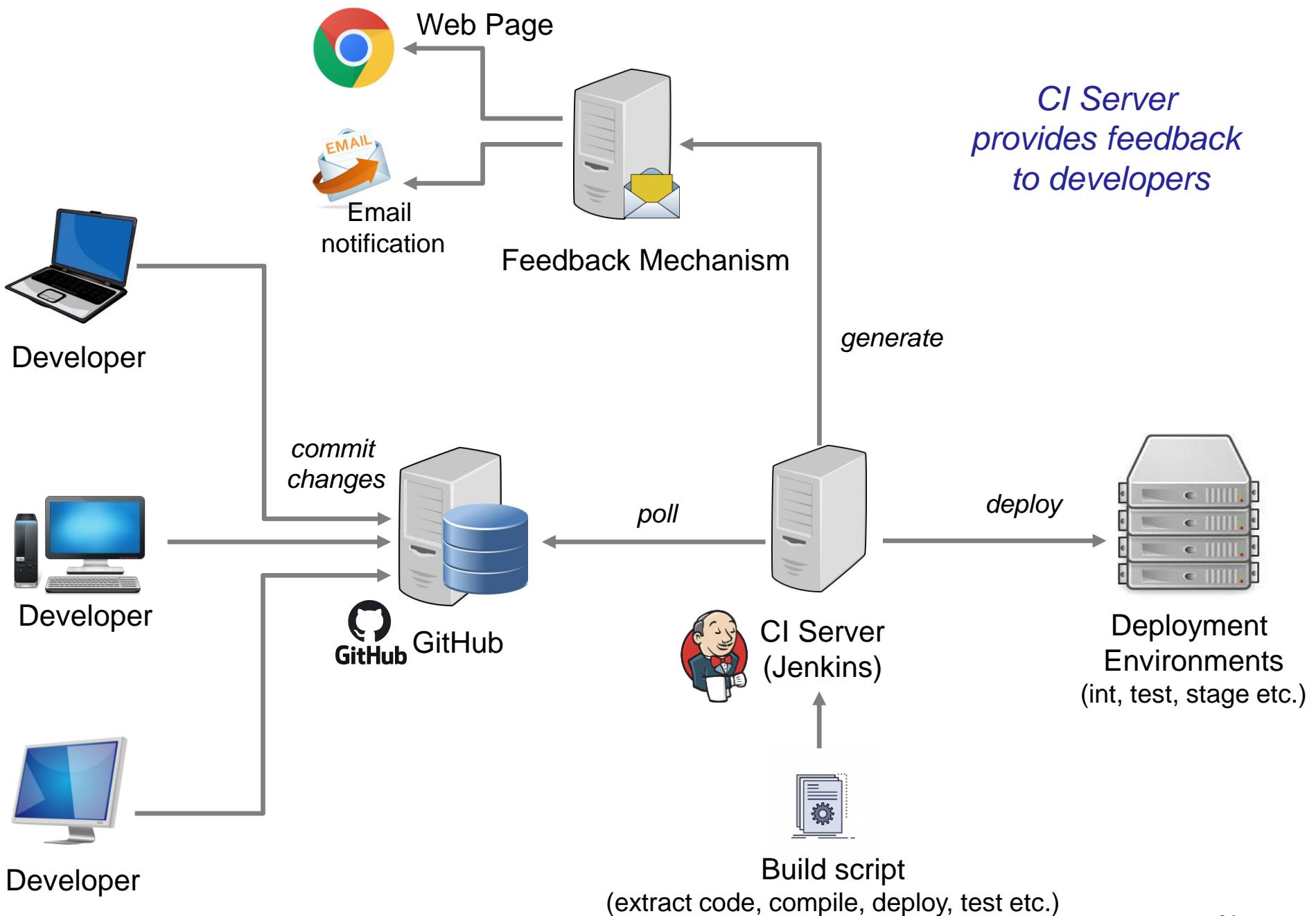


Continuous Integration Infrastructure



Continuous Integration Infrastructure





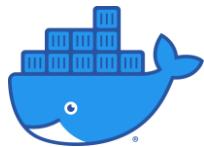
Continuous Deployment

Build Container Image

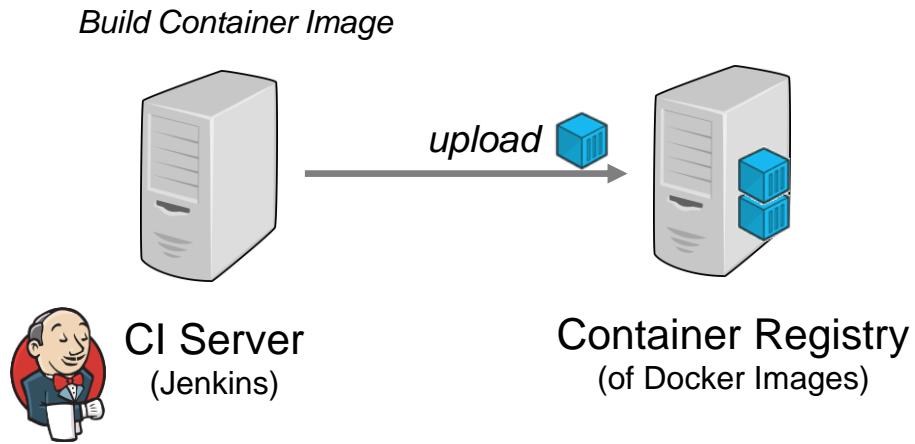


CI Server
(Jenkins)

- ❑ Jenkins builds Docker Container Images

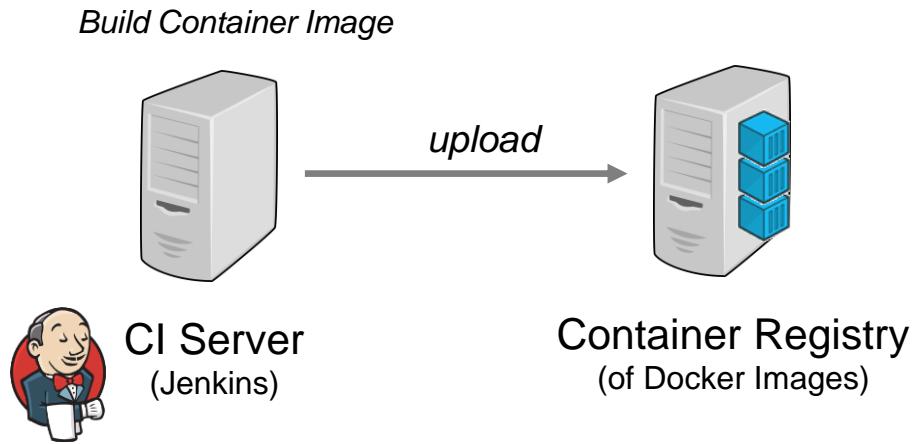


Continuous Deployment



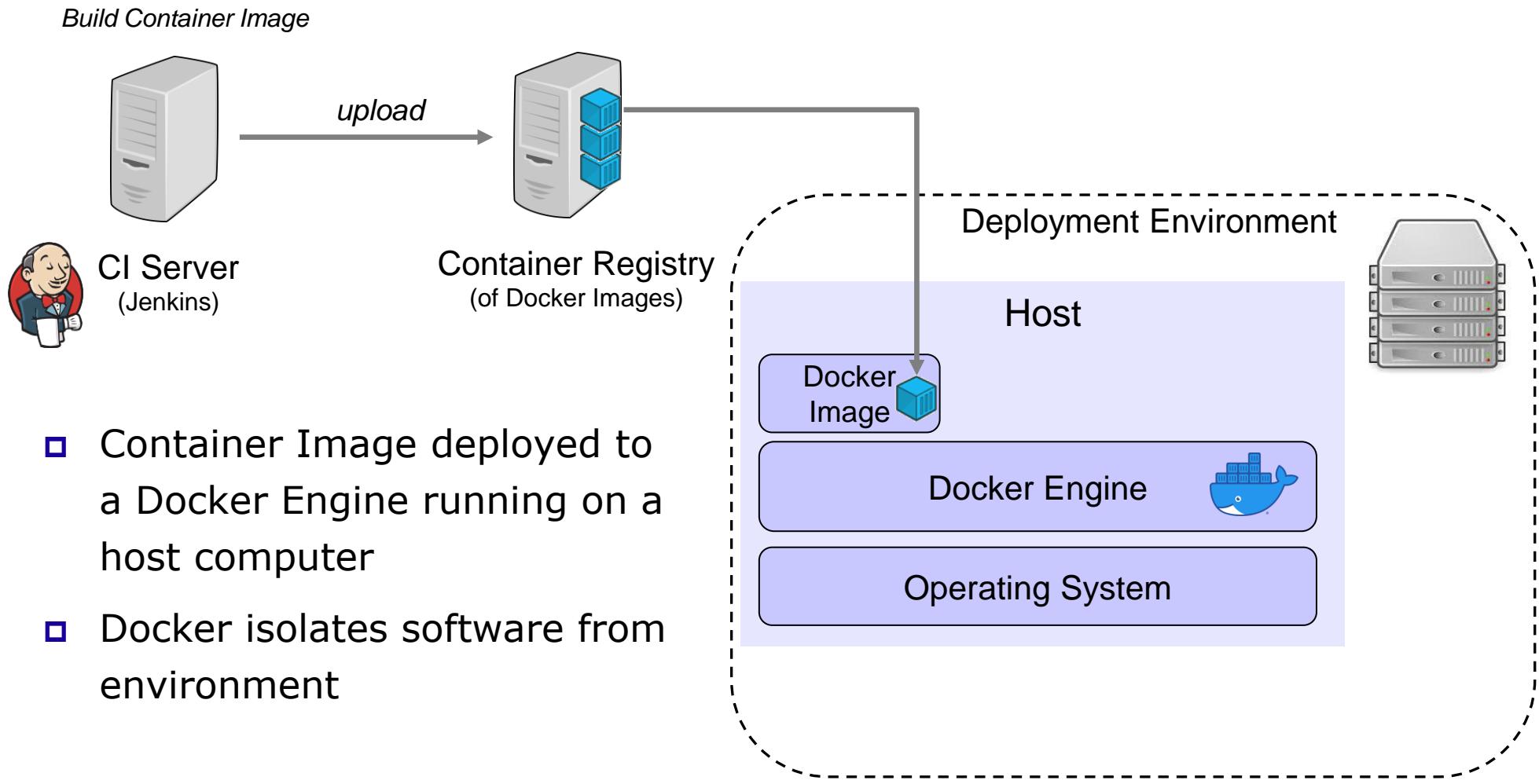
- Container images uploaded to Container Registry

Continuous Deployment

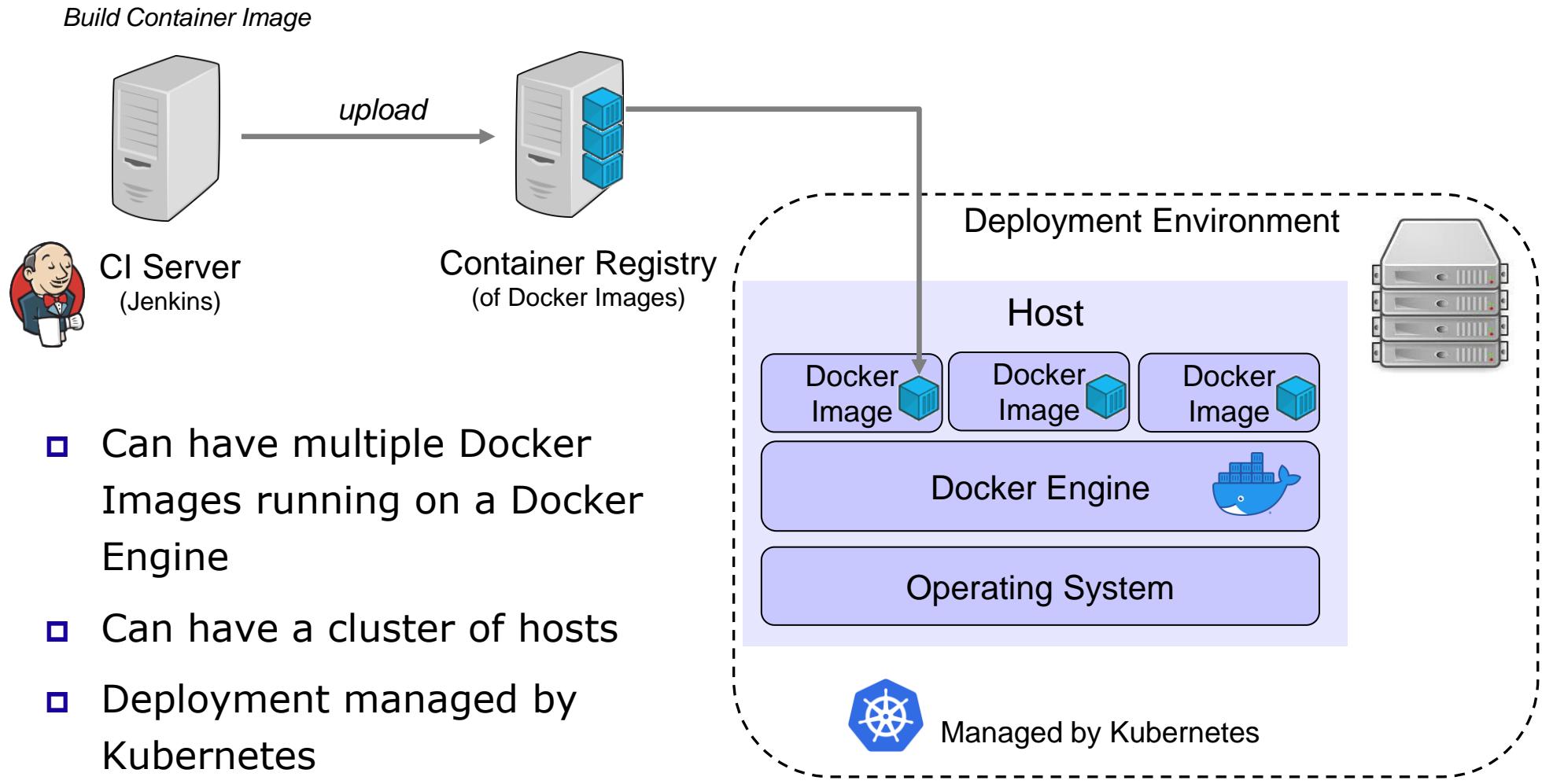


- Each image has everything needed to run app
 - code, runtime, libraries, settings, pictures, etc.
 - but not the OS so lighter weight than a Virtual Machine (VM)

Continuous Deployment

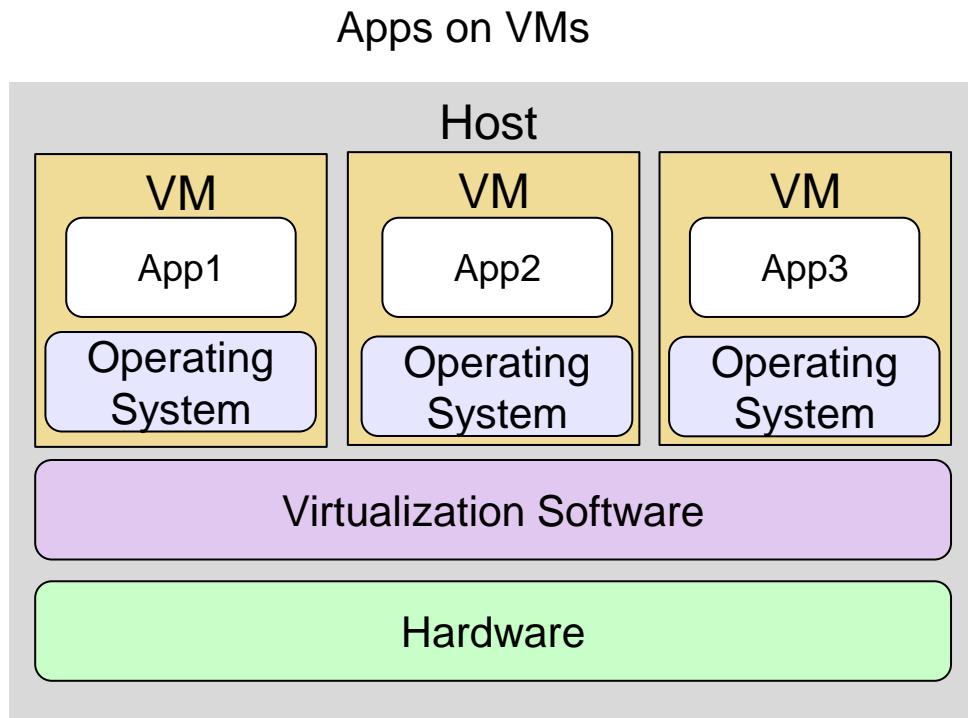
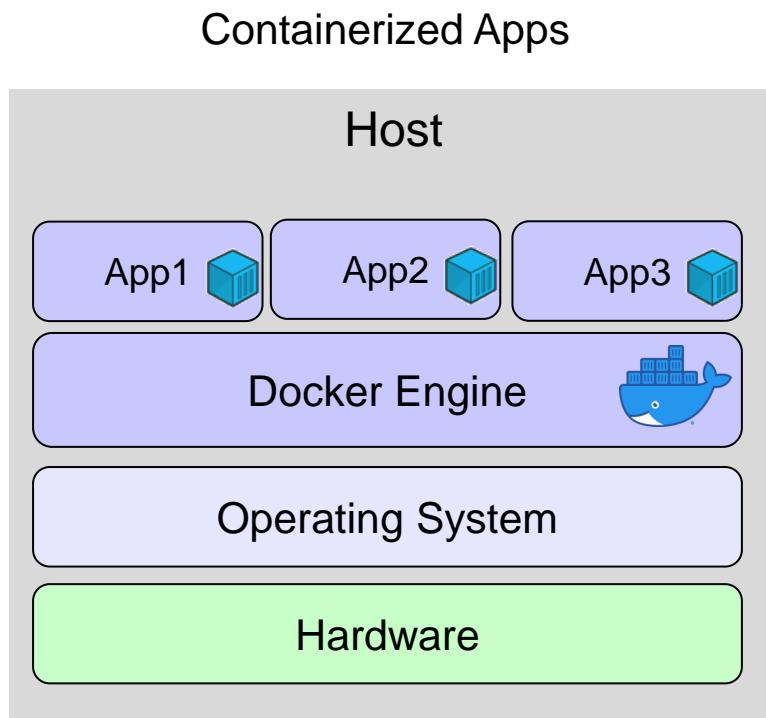


Continuous Deployment



Container v Virtual Machine

- Similar goals but very different operation



Developer Landscape

❑ JIRA for requirements tracking



The screenshot shows the JIRA web interface for the Jenkins project. The left sidebar has 'Issues' selected. The main area displays 'Open issues' with one issue highlighted: 'jenkins-cli.jar failed with java exception on LTS version 2.60.1'. The issue details show it's a bug, open, and assigned to Suresh Kumar. The URL in the browser is issues.jenkins-ci.org/projects/JENKINS/issues/JENKINS-46828?filter=allopenissues.

A web-based issue tracking tool

- add issues / watch / prioritise
- organise into projects / workstreams

How do you know what you should develop?

Developer Landscape



- Internal wiki for notes, documentation, discussion

Exxtreme Travel / Tours in Development

Rock Climbing in Colorado

Goals

Exxtreme Travel will be the first to provide a rock climbing tour offering in Colorado's Front Range. This is a great opportunity for us.

Background and strategic fit

Team rock climbing tours are one of our most popular offer categories and have done extremely well. Currently, none of our competitors have an offer on the market so we would be the first to offer such a tour.



Mia

Our customers love climbing as a team 😊

Jose

Yes, but we have to verify that it will be a good market fit for thrill seekers in Colorado.

Reply

≡

?

👤

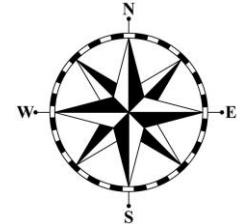
Git and GitHub/GitLab



Toby Dussek

Informed Academy





Plan for Session

- What is Git?
- Git Source Code Control
- Git Tools
- Command Line Git
- GitHub
- Git Change Flow
- File States in Git
- Adding Files in Git
- Working with a Remote Repository
- Branching

Git Introduction

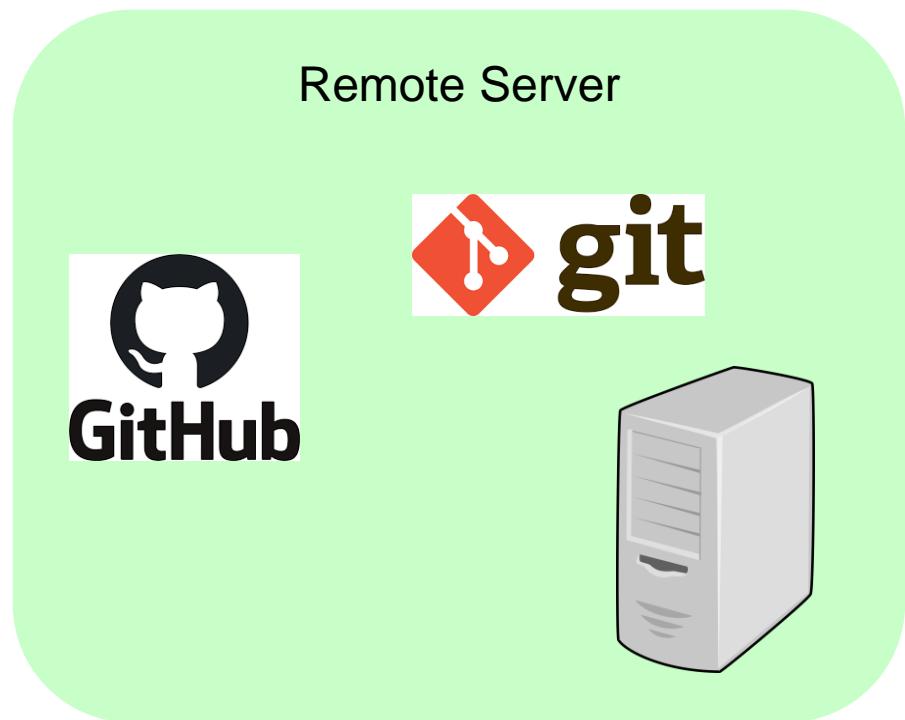
- Git is a
 - Source Code Configuration Management system
 - aka SCCM or SCM system
- Shared repository of code
 - to allow teams to work on codebase of a project
- Track software changes
 - so that those changes can be undone if necessary
- Tag releases
 - know exactly what code made it into a release
 - useful when different versions are needed for different clients

Git Introduction

- Developers store code locally and in a central repository



Local Client



Where do you store your code?

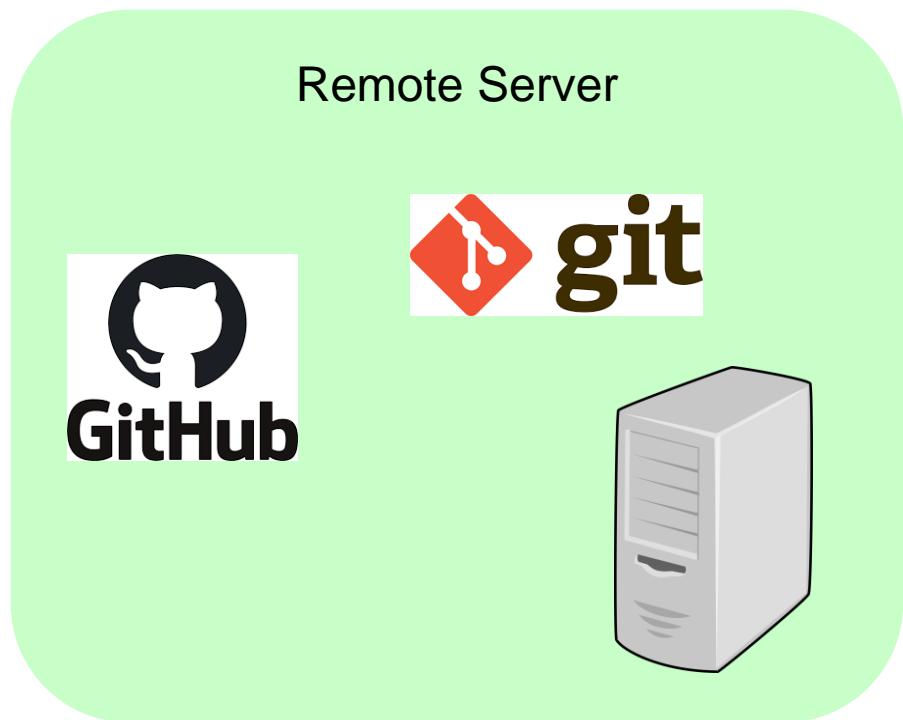
Git Introduction

- When they make changes they commit them to central repo



Local Client

*commit
changes*



Where do you store your code?

Git Introduction

- When they make changes they commit them to central repo



Local Client

*pull down
updates*

A diagram illustrating the relationship between a local client and a remote server. On the left, there is an icon of a computer setup labeled "Local Client". On the right, there is a large green rounded rectangle labeled "Remote Server". Inside the "Remote Server" box are the GitHub logo (a white octocat icon inside a dark blue square with the word "GitHub" below it), the Git logo (the red diamond and "git" text), and a grey server rack icon. A double-headed horizontal arrow connects the "Local Client" icon to the "Remote Server" box. Above this arrow, the text "pull down updates" is written in an italicized font.

Remote Server



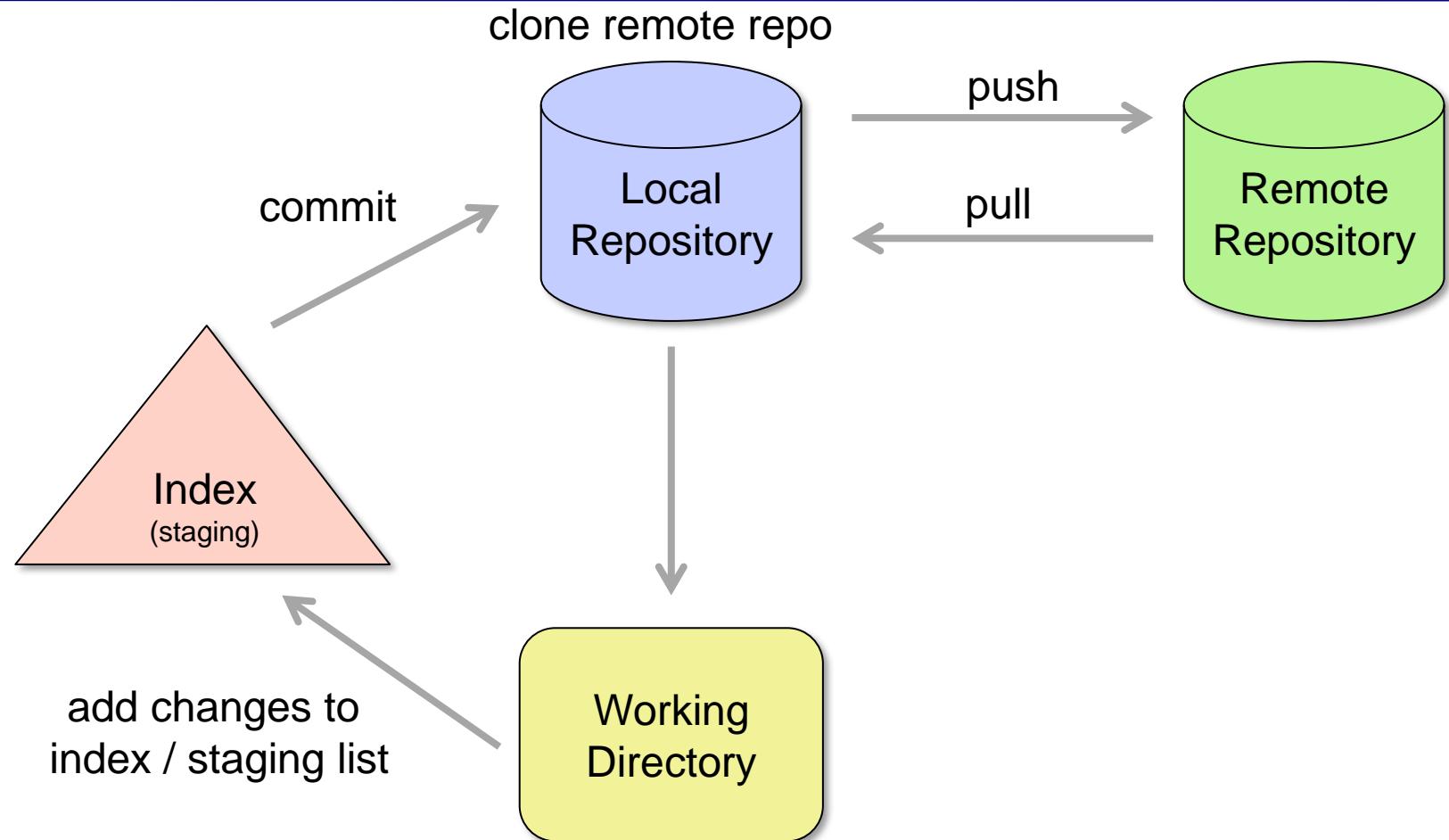
Where do you store your code?

Single Project Repository

- Commit to a shared repository
 - Allows code to be shared
 - during development, testing, maintenance
 - Allows you and others to check changes
 - But every check-in should have value
 - i.e. add something / improve something / fix something etc.
 - Don't leave it more than a day to check in your changes
 - All changes are checked in via a Pull Request
 - allows code to be reviewed
 - All assets are checked in
 - Database scripts, Java Code, Property files, JSON / XML files, JavaScript code, tests, test environment configuration etc.



Git Source Code Control



Git Tools

- Command Line Tools
 - basic command line interface for Git
- GitHub
 - web based interface to Git repos
 - supports Pull Requests (PRs) and other workflow options
 - can support reviews
 - diff tools etc.
- IDEs Git plugins / support
 - eGit for Eclipse IDE,
 - Git support built into IntelliJ
 - Git support also in Visual Studio Code etc.





Command Line Git

- Available for Windows, MacOS and Linux
- Can be used to create a personal project
 - can be useful for projects exploring how to solve problems
 - e.g. commit at regular intervals as you test new functionality
- If creating a new project from scratch, then
 - Need to initialize the new git project

```
$ mkdir my_project
$ cd my_project
$ git init
Initialized empty Git repository in my_project/.git/
$
```



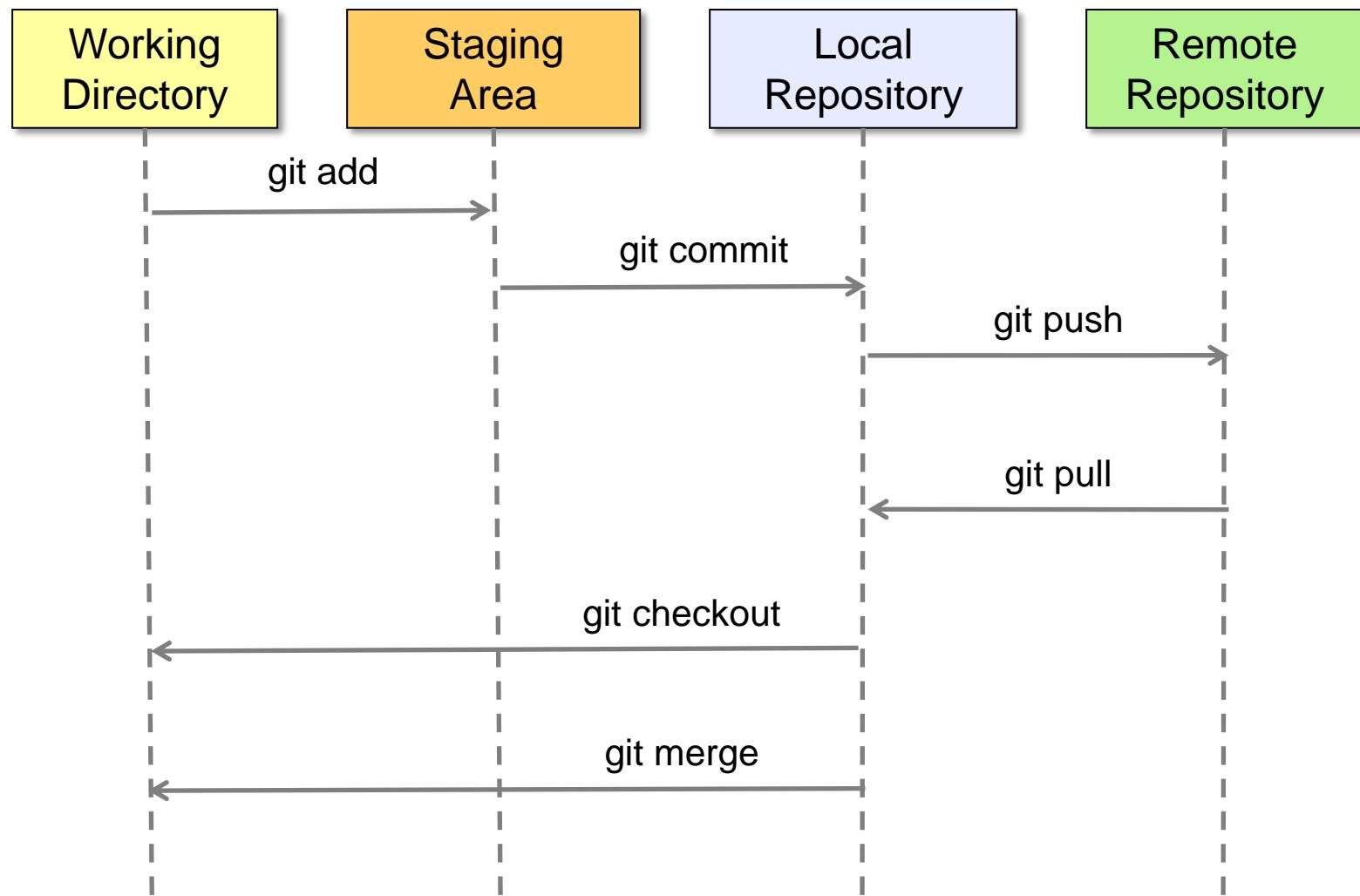
Command Line Git

- Can also clone an existing repository

```
$ git clone https://github.com/johnehunt/javascript-intro.git
Cloning into 'javascript-intro'...
remote: Enumerating objects: 8, done.
remote: Counting objects: 100% (8/8), done.
remote: Compressing objects: 100% (8/8), done.
remote: Total 7152 (delta 0), reused 4 (delta 0), pack-reused 7144
Receiving objects: 100% (7152/7152), 27.51 MiB | 7.71 MiB/s, done.
Resolving deltas: 100% (2495/2495), done.
Checking connectivity... done
$
```

- can now make changes to this repository
- add those changes to Gits change list (staging)
- commit them permanently into git repo
- push them to the GitHub hosted repo

Git Change Flow





File States in Git

- Files can be in one of four states

State	Description
untracked	file is not tracked by Git
staged	file is included in list of changes to be committed
tracked	file has been staged and then committed, Git tracks changes to this file
modified	file is tracked and has changed but not yet been staged



Adding Files in Git

- Need to stage changes
 - adds them to the list of changed files

```
$ cat > jjh-readme.txt  
Johns readme  
  
$ git add jjh-readme.txt
```

- Commit inserts the staged changes into the local repository

```
$ git commit -m 'My first commit'  
[master (root-commit) 6b0dfe6] My first commit  
 1 file changed, 1 insertion(+)  
  create mode 100644 jjh-readme.txt  
$
```



Working with Remote Repository

- Use the git push command to update the remote repository

```
$ git push
Username for 'https://github.com': johnehunt
Password for 'https://johnehunt@github.com': *****
Counting objects: 3, done.
Writing objects: 100% (3/3), 234 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/johnehunt/playarea.git
 * [new branch]      master -> master
$
```

- may need to specify the branch
 - e.g. git push master
- if a new branch
 - git push --set-upstream origin temp_branch



Branching

- Typically do development in a branch
 - Then merge your changes back into main branch
 - Typically via a Pull Request (or PR)
 - which has a second developer review the changes
 - and then pull those changes into the main branch
- Create branch using `git checkout -b <branch-name>`

```
$ git branch
* master
$ git checkout -b my_new_branch
Switched to a new branch 'my_new_branch'
$ git branch
  master
* my_new_branch
$
```



Branching

□ Master branch and my_new_branch

A screenshot of the GitHub branches page for the repository "johnehunt/playarea".

The page shows the following navigation and status indicators:

- Repository name: **johnehunt / playarea**
- Watched by 1 user
- Starred by 0 users
- Forked by 0 users
- Code tab selected
- Issues: 0
- Pull requests: 0
- Actions
- Projects: 0
- Wiki
- Security
- Insights
- Settings

The main content area has tabs for **Overview**, **Yours**, **Active**, **Stale**, and **All branches**. A search bar is available to "Search branches...".

Default branch

- master Updated 11 minutes ago by John Hunt
- Status: Default
- Change default branch button

Your branches

- my_new_branch Updated 44 seconds ago by John Hunt
 - 0 | 1
 - New pull request button
 - Delete button

Active branches

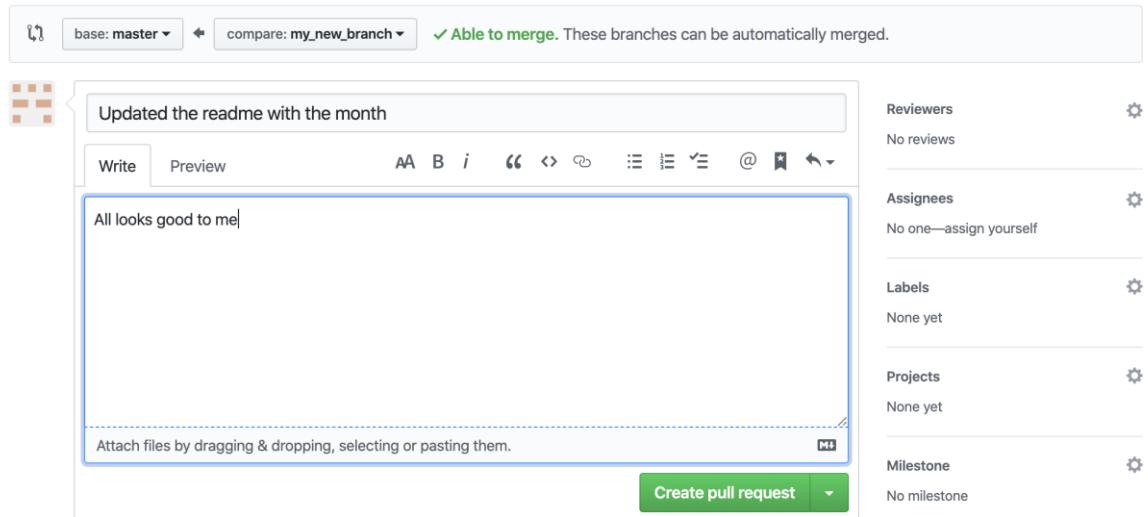
- my_new_branch Updated 44 seconds ago by John Hunt
 - 0 | 1
 - New pull request button
 - Delete button

Branching

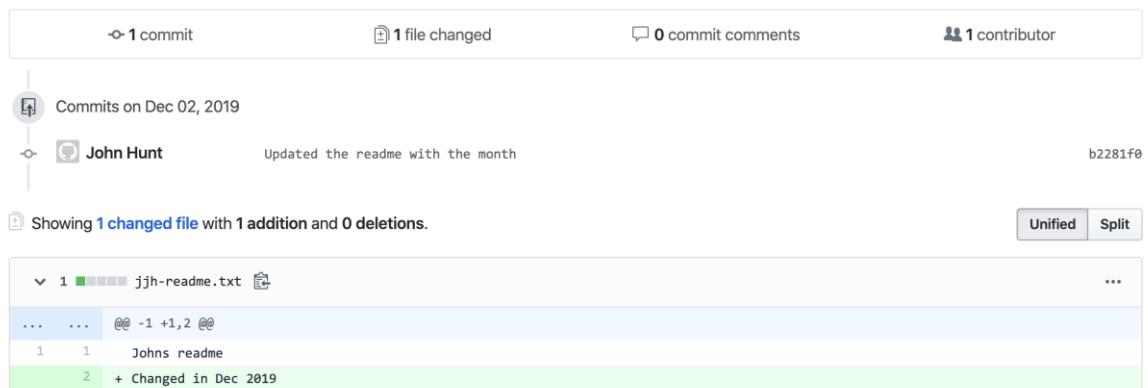
- Creating a PR (GitHub)
 - aka a Merge Request in GitLab

Open a pull request

Create a new pull request by comparing changes across two branches. If you need to, you can also [compare across forks](#).

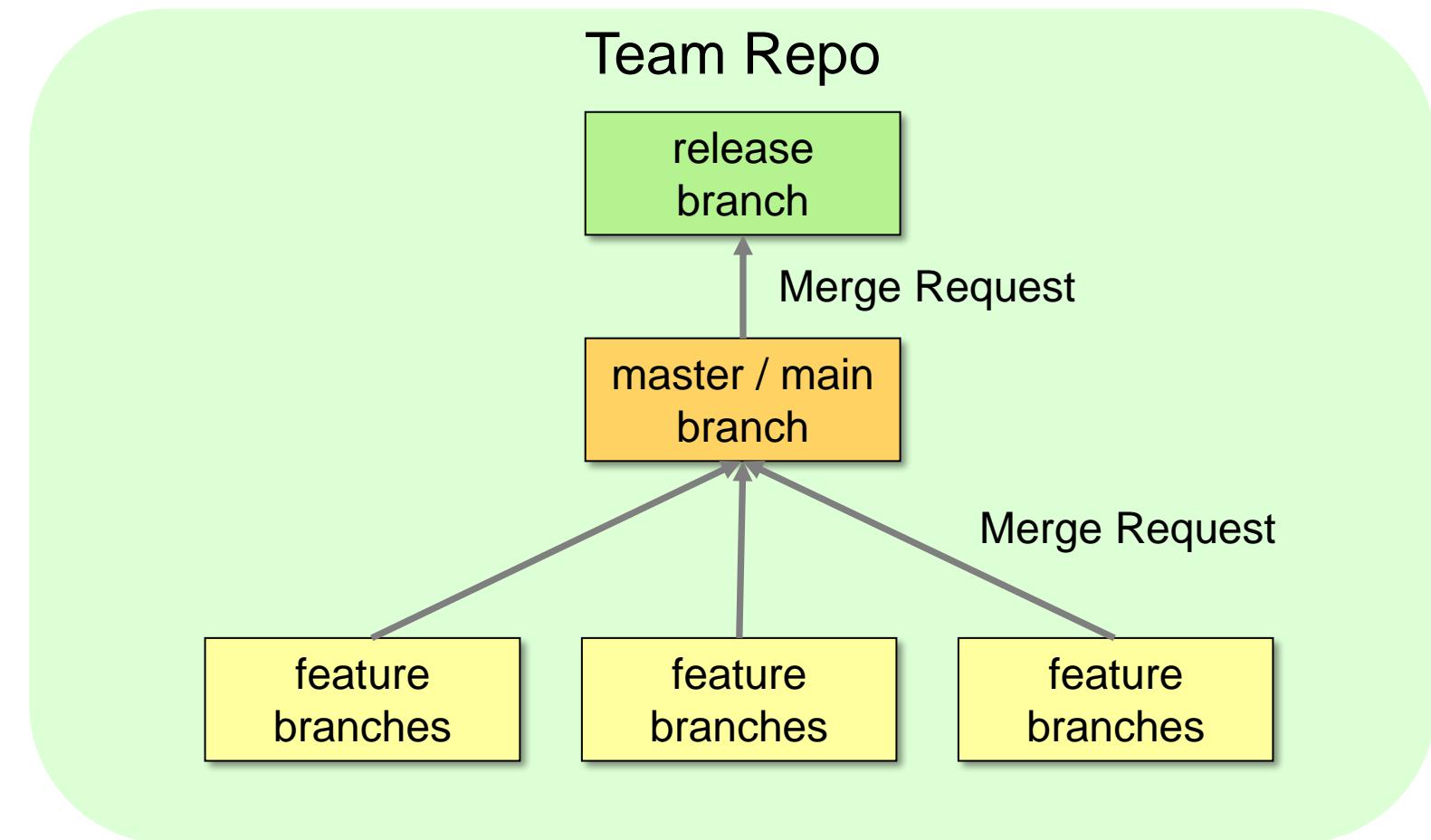


The screenshot shows the GitHub interface for creating a pull request. At the top, it says "base: master" and "compare: my_new_branch". A green checkmark indicates "Able to merge". The main area has a title "Updated the readme with the month" and a message "All looks good to me!". Below the message is a placeholder "Attach files by dragging & dropping, selecting or pasting them.". At the bottom right is a green "Create pull request" button.



The screenshot shows the GitHub pull request details page. It displays "1 commit", "1 file changed", "0 commit comments", and "1 contributor". The commit history shows "Commits on Dec 02, 2019" by "John Hunt" with the message "Updated the readme with the month". The file change summary shows "Showing 1 changed file with 1 addition and 0 deletions." The diff view shows the change in the "jjh-readme.txt" file, where line 1 was added with the content "Johns readme" and line 2 was added with the content "+ Changed in Dec 2019". There are buttons for "Unified" and "Split" views at the bottom right.

Multiple Branches



Exact terminology may differ from team to team

Git Terminology Quick Reference

Term	Description
HEAD	Symbolic reference to the latest version of the current branch
branch	reference to specific state of files in the working directory
staging area	Stores changes before they are committed (a change list) also called index
add	Add changes to the staging area
commit	Apply changes to the <i>local</i> repository
origin	The default upstream (remote) repository
push	Send changes from the local to the remote repository
pull	Retrieve changes from a remote repository and apply them to the local repository
fork syncing	Automated syncing from one repo to another
pull request	Request to merge two branches following review and approval process

Useful References

- Git Home Page
 - <http://git-scm.com>
 - <http://git-scm/doc> highly recommended
- Git Glossary
 - <https://www.kernel.org/pub/software/scm/git/docs/gitglossary.html>
- 10 Git Commands Every Developer should Know
 - <https://www.freecodecamp.org/news/10-important-git-commands-that-every-developer-should-know>
- Others
 - <http://www.kernel.org/pub/software/scm/git/docs>
 - <http://progit.org/book> Pro Git Book

Java Programs

Classes and Objects

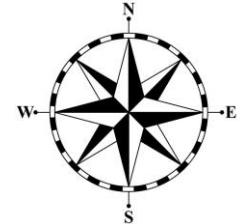


Toby Dussek

Informed Academy



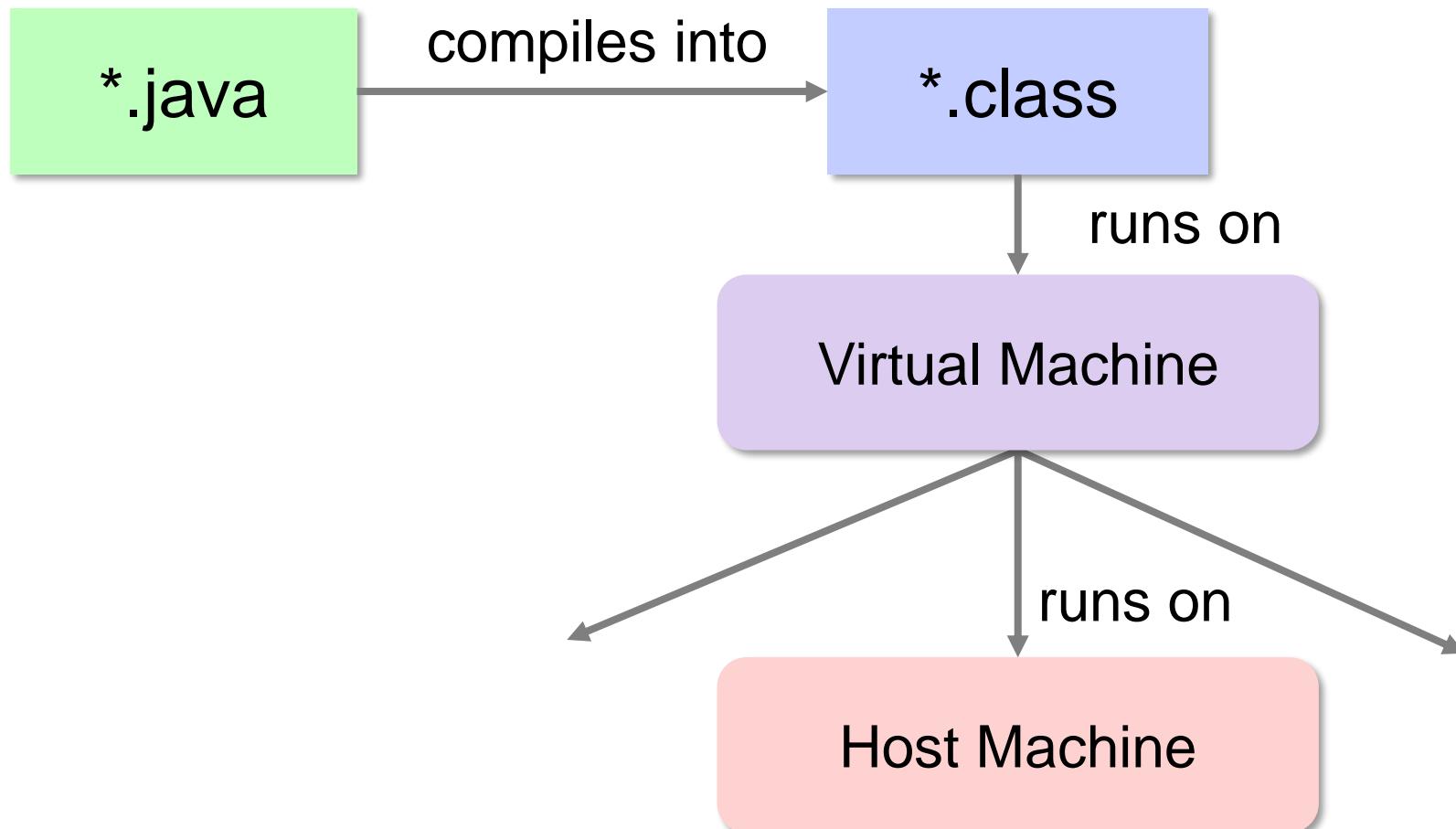
framework training
business value through education



Plan for Session

- How the Java Environment Works
- Hello World in Java
- What is a Class?
- Class Object Relationship
- The Person class
- Variable types
- Constructors
- Comments and Terminators
- Flow of Control and numeric operations

How the Java Environment Works



Hello World Program

□ Simple Hello World Program

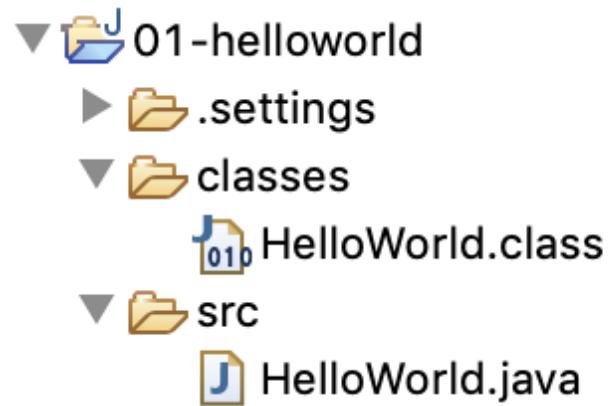
```
public class HelloWorld {  
  
    public static void main(String[] args) {  
        System.out.println("Hello World!");  
    }  
  
}
```

Hello World!

□ Compiled into .class file

- via javac
- can run via

```
java HelloWorld
```



The Person class

Person.java

```
public class Person {  
  
    // instance variables  
    private int age;  
    private String name;  
  
    // a constructor  
    public Person(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
}
```

Note:

- we have a constructor
- we have data local to any instance of Person
- instance variable declarations
- class Person defined in file Person.java
- this – it is a reference to the object itself

Java Types

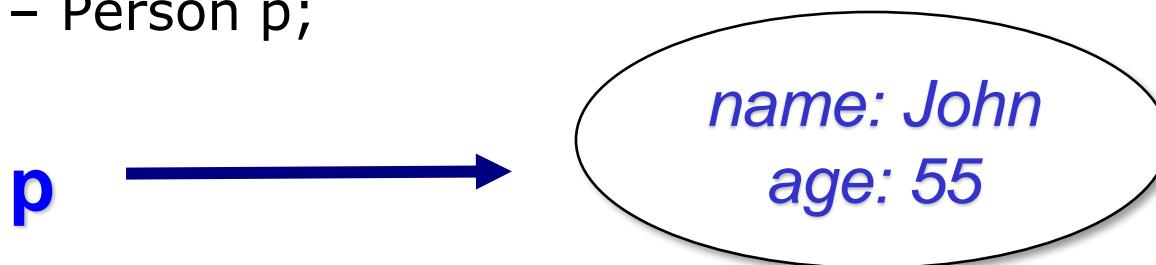
Actually Two types

- **Basic** / Fundamental types

- `int i;` or **boolean** flag;
 - `int i = 32;`

- **Reference** Types (objects / instances)

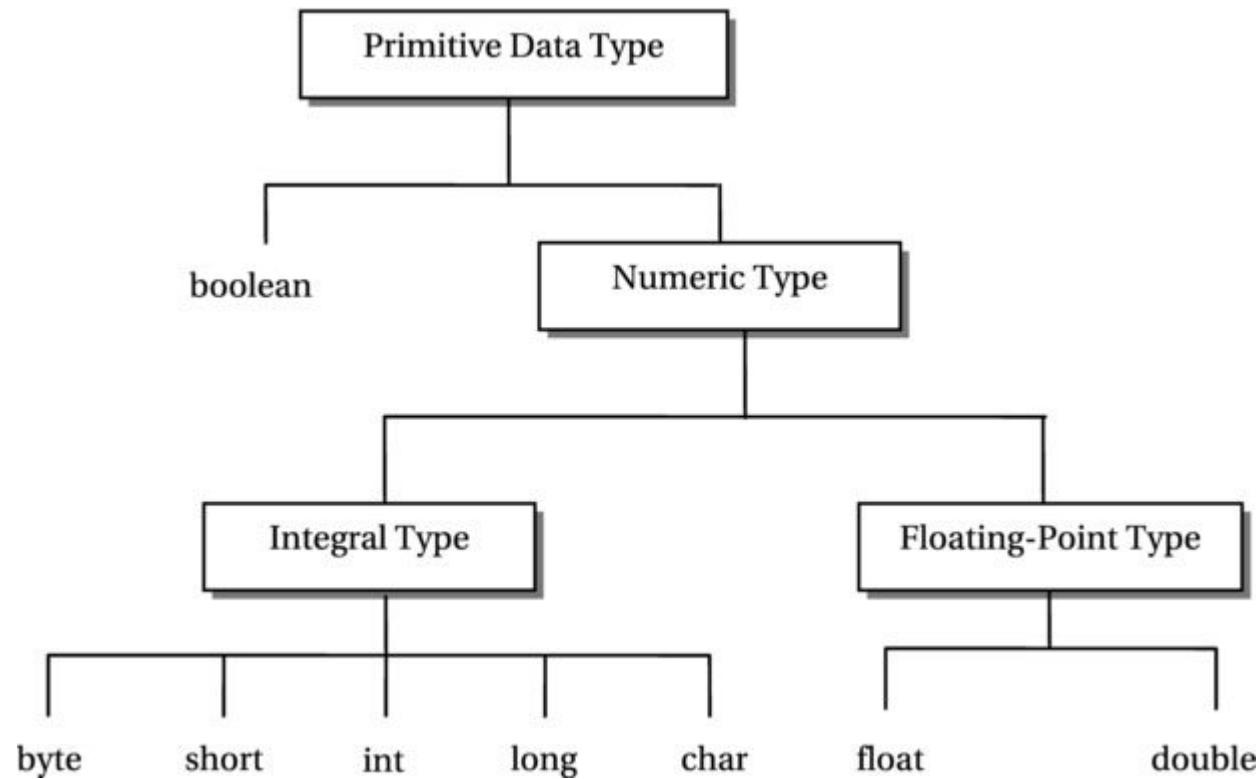
- types – Person p;



Java Variables

- All variables are typed
 - at compile time
- Variable declaration includes type
 - `String s;`
 - `int i;`
 - `double d;`
- Can be initialized when declared
 - `int i = 10;`
- Must be initialized before being referenced
 - `int i;`
 - `i = i + 1; // compile error as i is not initialized`

Java Data Types



Variable Types

- Instance variables
 - Defined for whole object
 - Initialized when instance created to null value
 - integrals - 0,
 - float - 0.0,
 - boolean - false,
 - char – '\u0000' character
 - Objects – null reference
- Local
 - Not initialized when created
 - Compiler will check when referenced

Constructors

- ❑ Have the following format:
 - ❑ used to *initialize* the *state* of the instance

```
public class ClassName {  
  
    public ClassName(.. parameters ..) {  
        ... statements ...  
    }  
  
}
```

- ❑ **Always** have at least one constructor
- ❑ Get default constructor (null Parameter)
- ❑ Can define your own (but lose default)

Overloaded Constructors

```
public class Person {  
  
    // instance variables  
    private int age;  
    private String name;  
  
    // a constructor  
    public Person(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
  
    public Person(String name) {  
        this(name, 0);  
    }  
  
    public Person() {  
        this("John");  
    }  
  
    // ... setters and getters omitted for brevity  
}
```

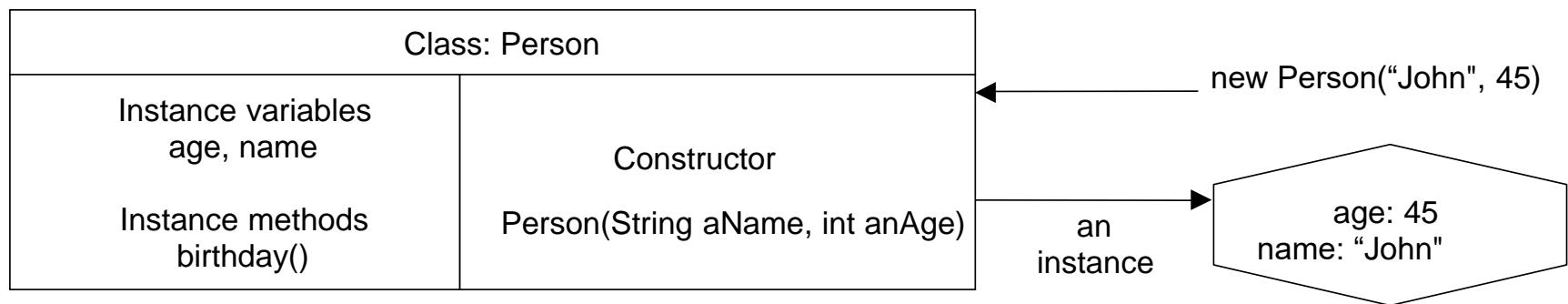
Using the class Person

- Fully functioning application
 - prints out default representation of a Person
 - note any public class must be in a file named after the class

```
PersonApp.java
public class PersonApp {
    public static void main(String[] args) {
        Person p = new Person("John", 54);
        System.out.println(p);
    }
}
```

Person@4b1210ee

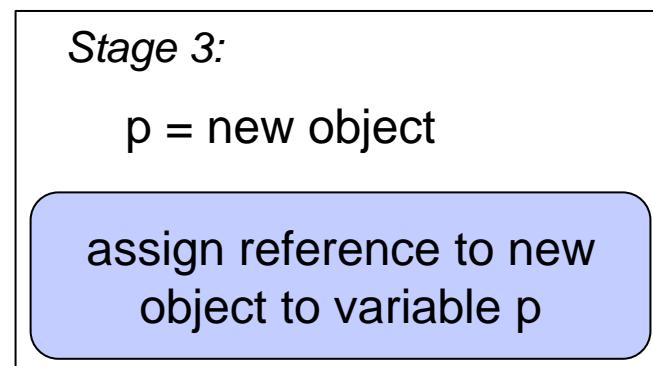
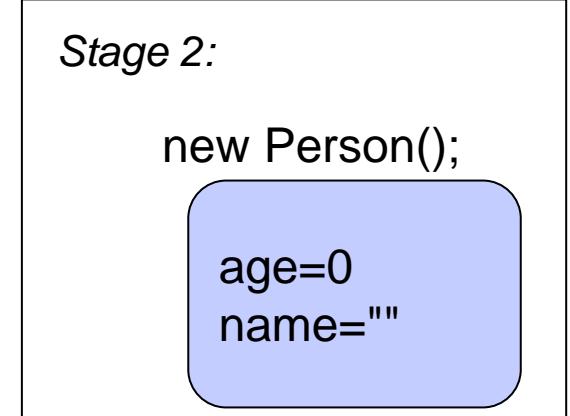
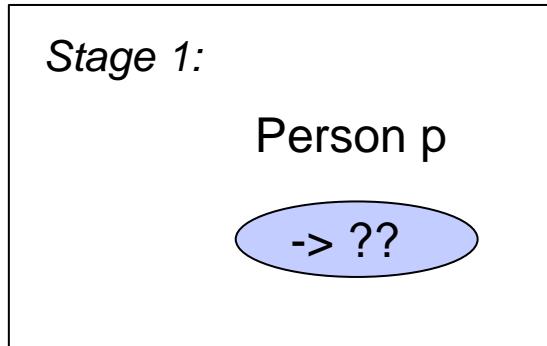
Instance Creation



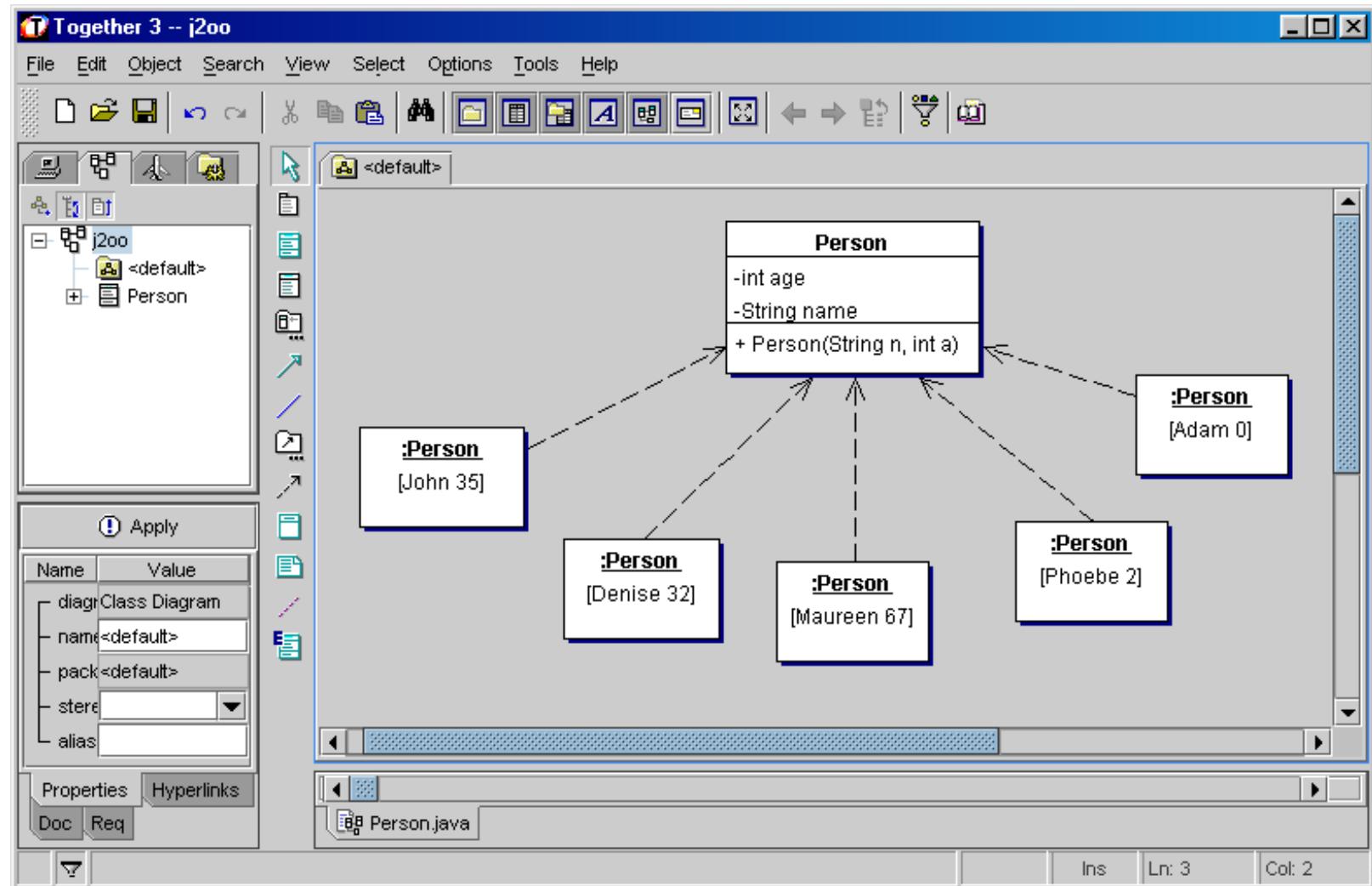
Instance Creation

- ❑ Actually a three-step process:

```
Person p = new Person();
```



Class versus Instances



Comments & Terminators

- Curly brackets define code blocks
- Semicolons are statement terminators:

```
{  
    int x;  
    x = 23;  
}
```

- Three types of Comment
 - // one line comment
 - /* multiple line comment */
 - /** javadoc comment */

Class Comments

Person.java

- `/** common convention */`
- Used by Javadoc tool to generate reference API material
- Used through Java itself to create online docs

```
/*
 * Class representing a Person object
 * <p>
 * <b>Status:</b> Draft
 * </p>
 * @author John Hunt Initial Development
 * @version 1.0
 */
public class Person {
    // instance variables
    private int age;
    private String name;

    /**
     * a constructor
     */
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

Javadoc tags

- Classes and interfaces

- @author @author John Hunt
- @version @version 3.2

- Methods and constructors

- @param @param temp the outside temperature
- @exception/@throws

- Methods

- @return @return the new value

- Others

- @see @see java.util.Vector
- @since @since 1.0 // of the system
- @deprecated / @serial / @link

Basic Java Types

- All common formats:

- byte, short, int, long – integer types
- boolean,
- float, double, - floating point numbers
- char

- Sizes are fixed:

- | | |
|--------------------------|----------------------------|
| □ boolean - true / false | char - 16-bit unsigned 'a' |
| □ byte - 8 bits | short - 16 bits (signed) |
| □ int - 32 bits | long - 64 bits |
| □ float - 32 bits | double - 64 bits |

- Numeric literals are int and double e.g.

- | | |
|-------------------------------------|----------------|
| □ 45 -> int | 27.3 -> double |
| □ (float f = 3.7; error pre Java 7) | |

Conditional Expressions

- If, while, for, switch, do - taken from 'C/C++' world

- if ($x == y$) { ... }
- while ($x < y$) {...}
- for (int $i = 0$; $i < 10$; $i++$) {...}
- do {...} while ($x >= y$)
- switch (i) { case 1:... break;....}
- {...} like begin / end
- Careful with equality: use $==$ instead of $=$
- for strings use `equals()` instead of $==$

If statement

- Condition must be a boolean
 - else optional

```
public class TestIf {  
    public static void main(String [] args) {  
        int i = 5, j = 10;  
        if (i < j) {  
            System.out.println(i);  
        } else {  
            System.out.println(j);  
        }  
    }  
}
```

5

- <cond>?<op1>:<op2>
Short cut conditional operator
 $x = (a < b) ? a : b;$

For statement

- Can have loop variables
- Can have multiple loop variables
- *i* is a loop variable

```
public class TestFor {  
  
    public static void main(String[] args) {  
        for (int i=0; i< 10; i++) {  
            System.out.println(i);  
        }  
    }  
}
```

0
1
2
3
4
5
6
7
8
9

Multiple loop variables

```
public class TestMultiFor {  
  
    public static void main(String[] args) {  
        for (int i=0,j=10; i<10; i++, j--) {  
            System.out.println(i + "\t" + j);  
        }  
    }  
}
```

0	10
1	9
2	8
3	7
4	6
5	5
6	4
7	3
8	2
9	1

While statement

- Test performed at start of loop
- Conditional must be a boolean

```
public class TestWhile {  
    public static void main(String[] args) {  
  
        int i = 5, j = 10;  
        while (i < j) {  
            System.out.println(i);  
            i++;  
        }  
    }  
}
```

5
6
7
8
9

Do statement

- Test performed at end of loop
- Always loops at least once

```
public class TestDo {  
  
    public static void main(String[] args) {  
        int i = 0, j = 10;  
        do {  
            System.out.println(i);  
            i++;  
        } while (i < j);  
  
    }  
}
```

0
1
2
3
4
5
6
7
8
9

Special flow of control operators

□ Allow some control of loops

- break (label); break or continue without a label
- continue (label); breaks the current loop
- label: <statement>

```
public class TestBreak {  
    public static void main(String[] args) {  
        int y = 10, x = 5;  
        loop: for (int i = 0; i < y; i++) {  
            for (int j = 0; j < x; j++) {  
                if (i == x) {  
                    break loop;  
                }  
                System.out.println(i + "\t" + j);  
            }  
        }  
    }  
}
```

Switch Statement

- Allows for multiple comparisons
 - must break out of case selection
 - otherwise will drop through
- Only supports subset of types
 - integers (byte, short, int – but not long) and object versions
 - char
 - enum (since Java 5)
 - String (since Java 7) uses equals() for comparison not ==
- Java 12/13 introduced experiment short cut version,

Switch Statement

```
public class SwitchStatementExample {  
  
    public static void main(String[] args) {  
        int value = 1;  
        switch (value) {  
            case 0:  
                System.out.println(0);  
                break;  
            case 1:  
                System.out.println(1);  
                break;  
            default:  
                System.out.println("Default");  
                break;  
        }  
    }  
}
```

1

Numeric Operations

- Basic operations
 - +, -, *, /
- Note that the result of + is always at least an **int**
 - byte b = (byte)c + (byte)d; // compile time error
- Two types of division operation
 - $7 / 2 \Rightarrow 3$
 - $7 \% 2 \Rightarrow 1$ // % is the modulus operator
- Also short cut operators
 - ++ // increment (pre and post fix)
 - -- // decrement (pre and post fix)
 - += e.g. $x += 2;$ // $x = x + 2;$
 - *=, /=, -=

Logical operators

- **And** && as well as &

&& - short cut **and**

```
if ((count != 0) && (total < 100) {...}
```

- **Or** represented by || as well as |

|| - short cut or

- ^ indicates XOR – exclusive OR

- ! Not operator

```
if (!flag) {...}
```

Printf Style Formatted Output

- Flexible string formatting
 - E.g. 30 cols for text and 2 for decimal:

```
String s = String.format("%30s %2d", name, age);
```

- Method signature for format method:

```
public static format(String format, Object... args) {...}
```
- Convenience method on PrintStream class:

```
System.out.printf("%30s %2d", name, age);
```

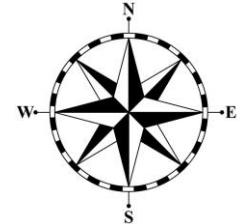
Maven: Dependency Management and Project Builds



Toby Dussek

Informed Academy





Plan for Session

- Project Builds
- Maven 2.0
- Project Object Model (POM)
- Project Structure
- Maven Dependency Management
- Maven Lifecycle phases & goals
- Maven commands

Project Builds

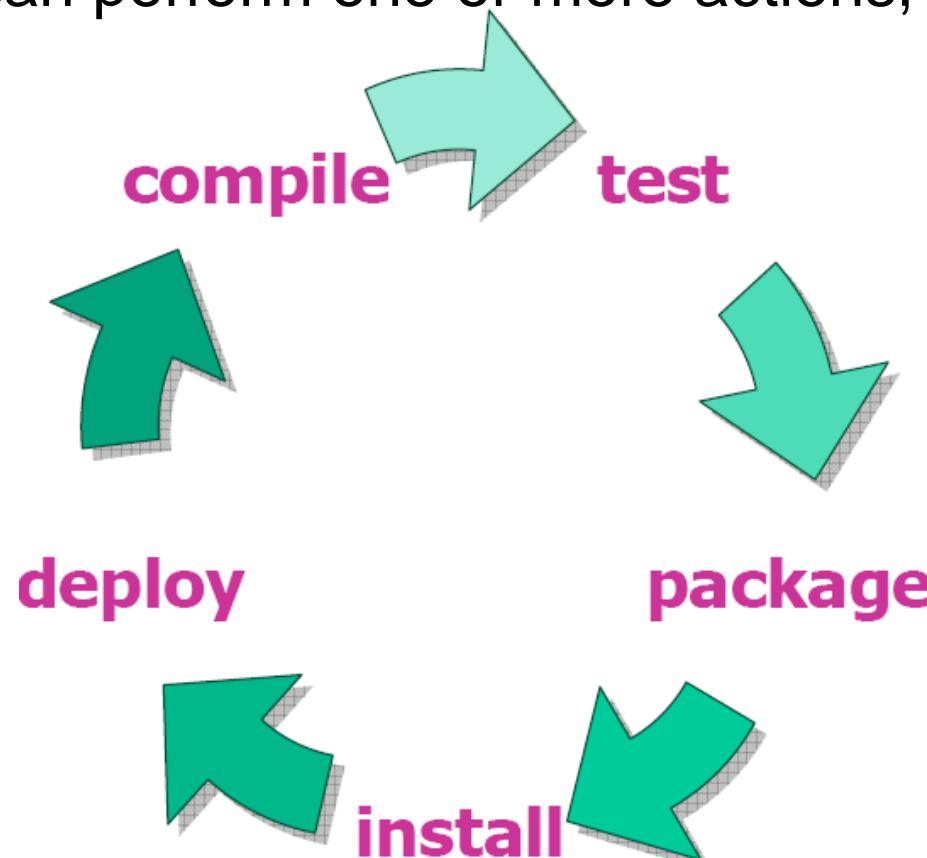
- Applications are made up of multiple components
 - HTML and CSS files and image files
 - JavaScript files and libraries
 - Java / Python / JavaScript etc. services
 - Libraries used by Java / Python / JavaScript
 - XML / JSON / YAML based configuration files
 - Database scripts
 - Property or metadata data files
- Also building a project might require several steps or involve different phases

Project Build Tools

- Can build your projects manually
 - Tedious and error prone
- Can use IDEs like Eclipse & IntelliJ
 - Easy, but not very portable to server environments
- Can write scripts to automate the process using tools like ant
 - Good, but much time spent on designing and testing scripts
- Maven comes with useful internal knowledge
- What a project is
 - How to build it
 - A build life cycle

Project Build Cycle

- The build life cycle consists of a series of phases where each phase can perform one or more actions, or goals



Maven 2.0

- Industry standard build tool
 - Understands project lifecycle as well as goals
 - <http://maven.apache.org/> (The maven site)
- Handles dependencies with libraries
 - Downloaded from a repository
- Uses concept of *convention over configuration*
- Can also handle versioning of builds
- Also has concept of plugins to extend Maven
 - <http://www.codehaus.org/> (Add ons for maven)

Project Object Model

- Core concept in Maven is the POM
 - aka Project Object Model file
- The POM contains detailed METADATA information about the project:
 - Versioning and configuration management
 - Dependencies
 - Project type e.g. jar (default), war, ear
 - Application and testing resources
- And potentially much more ...
 - But not often required

Example POM File

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" ...>

    <modelVersion>4.0.0</modelVersion>

    <groupId>com.jjh</groupId>
    <artifactId>hello</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>hello</name>
    <url>http://www.midmarsh.com</url>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <maven.compiler.source>11</maven.compiler.source>
        <maven.compiler.target>11</maven.compiler.target>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.junit.jupiter</groupId>
            <artifactId>junit-jupiter-api</artifactId>
            <version>5.7.0</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
</project>
```

How the POM can work

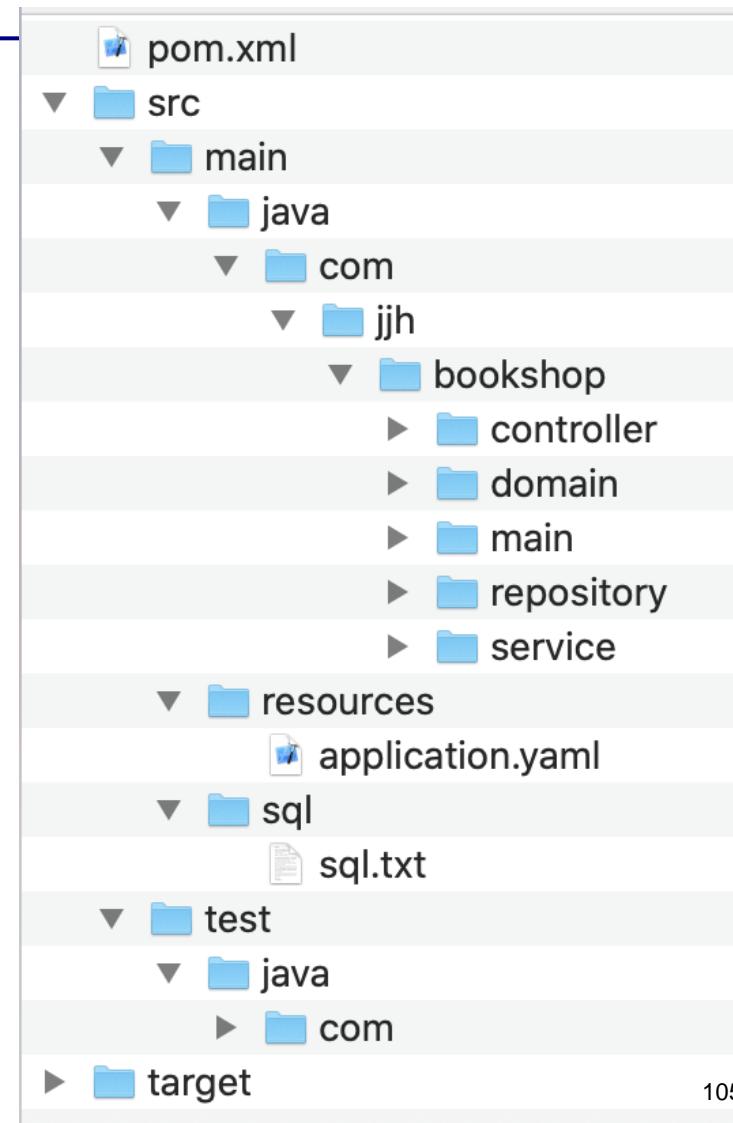
- Super POM is parent of all POMs
- Maven implicitly uses its Super POM
- Super POM carries with it all the default conventions that Maven encourages
- Similar in concept to a base class

POM Concepts

- Parent-Child relationships
 - Represented by the parent tag
- Library dependencies
 - Represented by the dependency tag
- Project templates
 - Represented by the archetype plug-in
- Properties
 - Key/Value pairs used to provide values used in rest of POM

Default Project Structure

- Based on type of project
- Note in this project
 - src/main/java
 - src/main/resources
 - src/test/java
 - src/test/resources
- Common patterns
- *target* contains class files



Archetypes

- Define default settings / configuration for common types of projects
 - for example web, enterprise, command line
 - but also specific technologies such as
 - basic Java app, web app, Spring Boot, mobile app etc.
- Archetypes typically include
 - default dependencies
 - project structure
 - lifecycle steps
 - initial application / initial set of tests
- See effect of web application archetype on pom.xml

Maven Dependency Mechanism

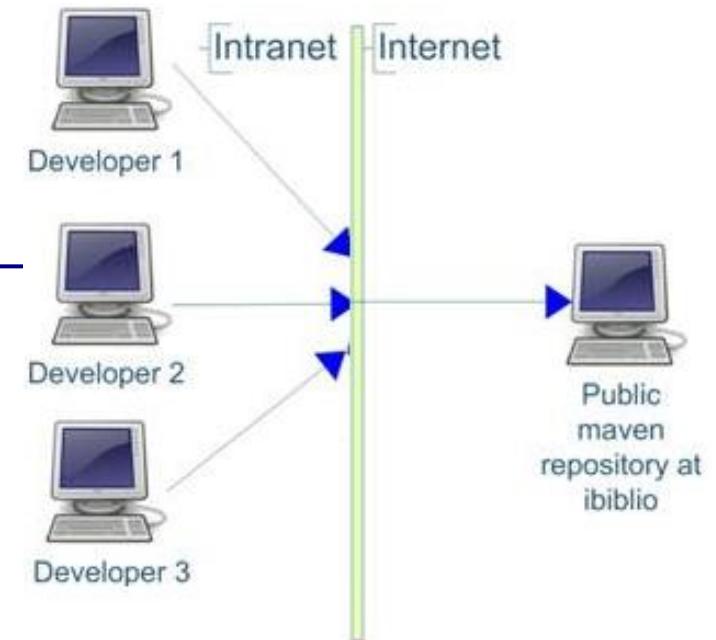
- Define your dependencies
 - Don't include jar file

- Maven will handle:
 - Downloading the dependencies
 - Resolving all transient dependencies
 - Adding dependencies to compilation and test classpath/include path
 - Bundling dependencies when needed

Maven Repositories

There are two types of repositories:

- Two types of repository:
 - **Remote**, accessed via a variety of protocols
 - Central one at <http://repo1.maven.org/maven2/>
 - **Local**, cache of downloaded artefacts and latest builds
 - Local - a local cache of the remote downloads and the latest build artifacts
 - Default is \${user.home}/.m2/repository
- Plus local (organisation) Intranet repository



Maven Dependencies

- List groupId, artifactId optionally version & scope
- Can also specify additional repositories to look in

```
<dependencies>
  <dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-api</artifactId>
    <version>5.7.0</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-engine</artifactId>
    <version>5.7.0</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-params</artifactId>
    <version>5.7.0</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

Maven Dependency Scope

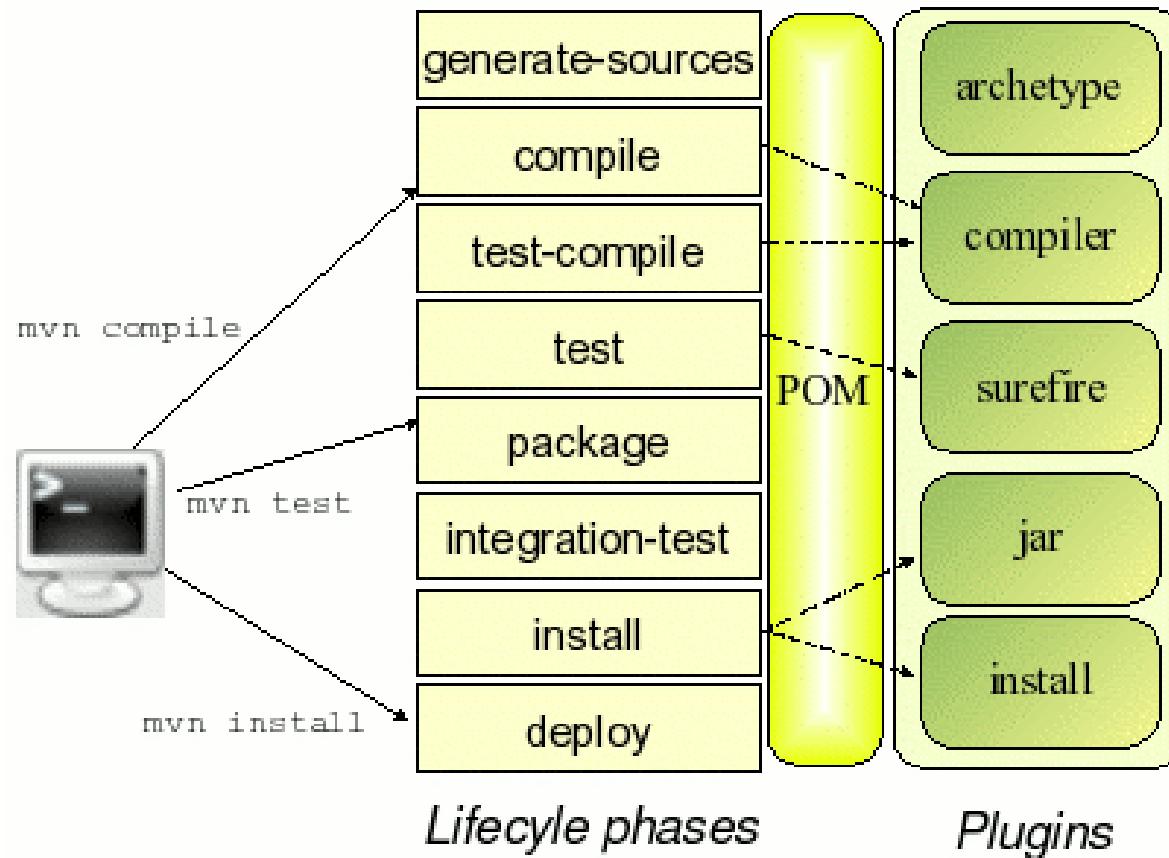
- Primarily used to affect the classpath
- There are 6 scopes available:
 - **compile**
The default. Dependencies are available in all classpaths
 - **provided**
Indicates that the JDK or a container will provide the dependency at runtime but must be obtained at compile time.
 - **runtime**
Dependency only added to runtime and test classpaths.
 - **test**
Only available for the test compilation and execution phases.
 - **system**
Like provided but **you** provide the jar file

Maven lifecycle

- Maven has concept of a build lifecycle
 - E.g. clean, build, etc.
- Build Lifecycle is Made Up of Phases
 - **validate** - validate the project is correct
 - **compile** - compile the source code of the project
 - **test** – run tests associated with the project
 - **package** - package as appropriate (e.g. jar, war)
 - **install** - install the package into the local repository
 - **deploy** - copies the final package to the remote repository
- Plugins can extend this
 - E.g. Tomcat Maven plugin used to deploy to Tomcat

Phases and Plugins

- Can invoke phases
 - Maven works out what is required to meet the phases requirements



Common Maven Commands

- Format of all maven commands is
 - mvn <command> <options>

- For example
 - mvn –version
 - mvn clean – remove all build files
 - mvn compile – compile the application
 - mvn package – do what is needed to package the app
 - mvn test – run any tests
 - mvn install – install to local maven repo

Combining phases together

- Can specify a plugin and then the goal
 - mvn jar:jar use the jar plugin to create a jar file
 - mvn resources:resources – copy all resources
 - mvn eclipse:eclipse – build eclipse files
 - mvn dependency:tree – generate dependency tree

- Can combine phases and goals together
 - mvn clean resources:resources package
 - mvn clean deploy

References

- Online
 - <http://maven.apache.org/>
 - <http://maven.apache.org/guides/getting-started/maven-in-five-minutes.html>
 - <https://www.baeldung.com/maven>

Roles in Software Development

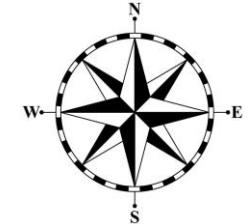


Toby Dussek

Informed Academy



framework training
business value through education



Plan for Session

- Roles in Software Development
- Summary of Selection of Roles
- Management Roles
- User / Requirement Related Roles
- Technical Roles
- Other Roles
- Release Management



Roles in Software Development

- Many different roles
- Differ from organisation to organisation
- In a small organisation an individual may play many different roles
- In a large organisation roles may be clearly demarcated
- Also several stakeholders
 - those who have additional involvement in project
- Each have their own responsibilities and areas

What Roles are there?



Summary of Selection of Roles

- Project Sponsor
- Subject Matter Experts (SMEs)
- Product Owner
- Project Manager (PM)
- Business Analyst
- User Researcher
- Interaction Designer / Usability Designer
- Content Designer
- Software Architect
- Technical Lead
- Service Designer
- Software Developer
- Software Tester
- User Acceptance Tester
- DevOps Engineer

Management Roles

- Project Sponsor / Business Unit Manager
 - provides direction and (typically) sign off on financial resources for project
 - provides an escalation path for the Product Owner
 - may also handle changes in scope; go/no-go decisions
- Product Owner / Product Manager
 - represents business or end users to project team
 - helps determine what feature will be in each release
 - helps to prioritise backlog
 - helps to gather requirements / user stories
 - ensures product vision is adhered to

Management Roles

- Project Manager
 - manages the s/w project
 - has overall responsibility for all aspects of the project
- Scrum Master
 - ensure everyone follows Scrum practices
 - acts as Scrum coach
 - removes impediments so team can progress
 - holds daily Stand-Up
 - holds end of Spring retrospective
 - organises start of Sprint planning meeting

User / Requirements Related

- SMEs
 - Subject Matter Expert
 - Domain Expert related to project
 - Know their roles inside out
 - help to verify, clarify, confirm operation needed from the system
- Business Analyst
 - work to refine and define / clarify required features
 - sit between the business and the dev team
 - speak the language of both worlds

User / Requirements Related

□ User Researcher

- focusses on understanding the user
- their behaviours, needs and motivations
- through observation, task analysis, interviews, role play etc.
- aims to inform and support content development and UI design

□ Content Designer

- designs content included in app
- e.g. video, text, images, colours,
- need to know their audience
- can work on the web, in print or in digital services & apps

User / Requirements Related

- UX Designer / Interaction Designer
 - develop visual look and feel of product
 - develop UI standards to be adhered to
 - ensure suitable User Experience (UX)
 - determine flow through product
- Service Designer
 - still within user space
 - work to design the *service* required by the end user
 - determine the features needed
 - explore how the *service* can be supported by the system

Technical Roles

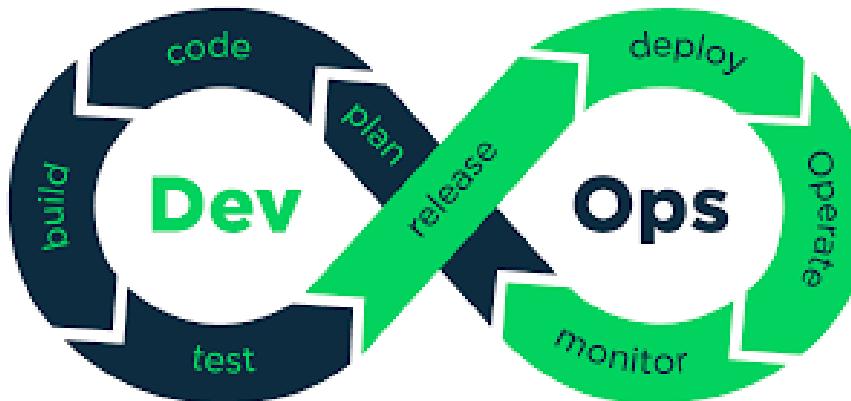
- Software Architect
 - identifies / designs the overall architecture of the system
 - end-to-end, cross functional system design
 - communicating architecture to others e.g. developers
 - testing architectural elements of the project
- Technical Lead
 - aka development team leader
 - translates business requirements into technical solution
 - helps to produce estimates
 - establishes and enforces standards, processes, practices
 - ensures developers work within architectural framework

Technical Roles

- Software Developer
 - front-end, back-end, database etc.
 - take technical requirements and implement technical solutions and associated tests
 - peer review others work
 - feed into estimation process
- Software Tester
 - focus on the process of testing the system
 - may be focussed on specific types of testing, integration, system, performance, scalability etc.
 - are responsible for executing tests / reviewing test results
 - developers in test often used with Agile methods

Further Roles

- User Acceptance Tester
 - focussed on User Acceptance
 - often work closely with SMEs
 - also on other Quality Assurance (QA) criteria
 - such as usability
- DevOps Engineer



Release Management

- Software products (particularly web apps) are
 - in an ongoing cycle of development, testing and release
 - often running on evolving platforms and frameworks
 - with growing complexity
- Such systems require
 - dedicated resources to oversee the integration and flow of development testing, deployment, maintenance and support
- Release Management is process of
 - managing, planning, scheduling and controlling software build
 - through different stages and environments
 - inc. testing and deploying software releases

Release Management



Working with Methods

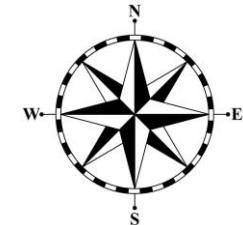


Toby Dussek

Informed Academy



Plan for Session



- Method Definitions
- Method Arguments & Return Types
- Overloading Methods
- Method Invocations
- Defining Methods
- Constructors Versus Methods
- `toString()` method
- Adding behaviour to the class Person
- The PersonApp program
- Annotations

Method Definitions

- Have the following format

```
access-modifier returnType methodName (args) {  
    local variable definitions  
    statements  
}
```

- For example

```
public void birthday() {  
    age = age + 1;  
}
```

- Are inherited by subclasses
- Return value using **return** operator
 - `return 42;`

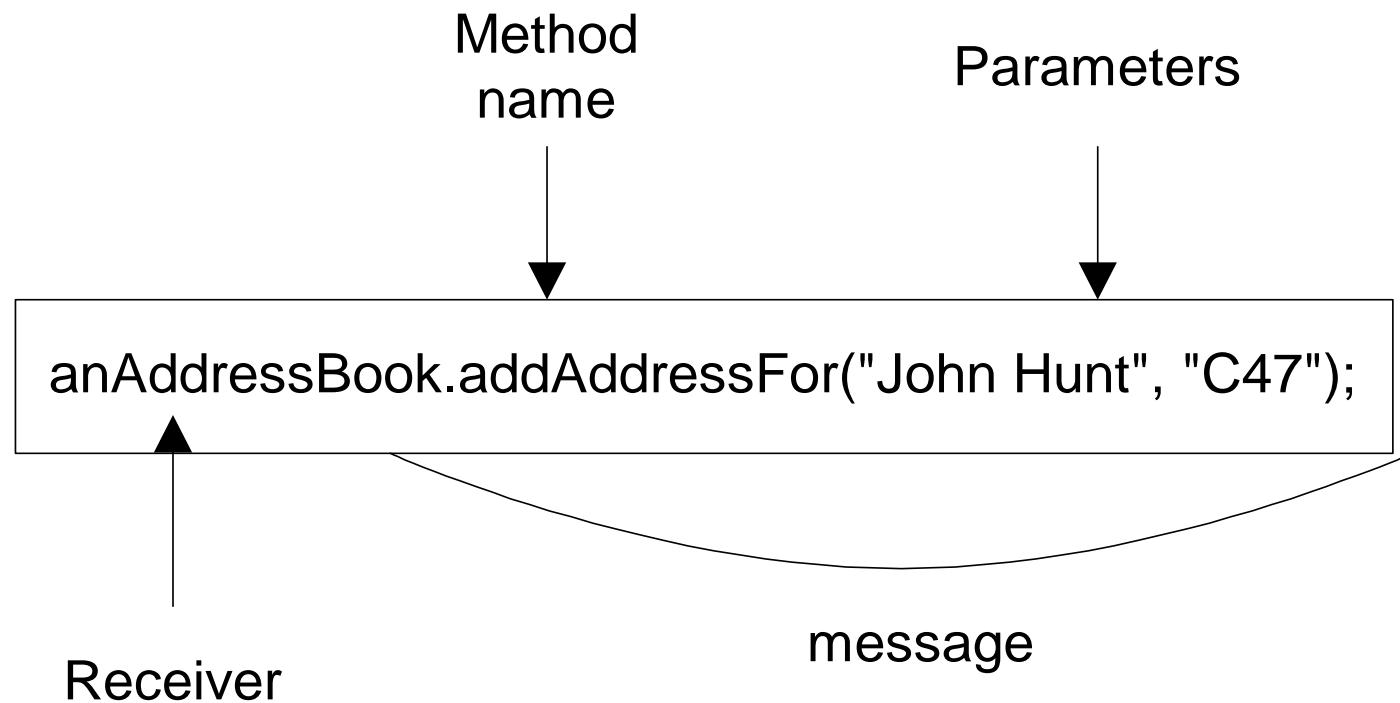
Method Arguments & Return Types

- Each argument must have its own type
 - Even if they are the same type
 - Separated by comma
 - e.g. **public void** add(String name, String address) { ... }
- Every method must have a return type
 - Even if it is **void**
 - Returned value must be of specified type or “castable” to the type
- Method selected by name *and* arguments

Overloading Methods

- Can have more than one method with the same name
 - e.g. `print()` in class **java.io.PrintWriter**
- Different methods distinguished by parameters
 - e.g. `print(String)`, `print(int)`, `print(double)`
- Rules:
 - Argument list must differ (note automatic casting)
 - Return types can differ but not sufficient to differentiate
 - Visibility can differ

Method Invocation



Defining Methods

- Extending Person to have some methods:
 - getAge returns an int
 - setAge takes an int as a parameter and returns void

```
public class Person {  
  
    // instance variables  
    private int age;  
    private String name;  
  
    // a constructor  
    public Person(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
  
    public int getAge() {  
        return age;  
    }  
  
    public void setAge(int age) {  
        this.age = age;  
    }  
}
```

Constructors Versus Methods

- Always at least one (Default) constructor
- Not method as
 - Can't call directly
 - Have no return type
 - Must have the same name as the class
 - Are not inherited
- Both can be overloaded
 - Person() & Person(String name)
 - setAge(int age) & setAge(String age)
- Use "this" to call one constructor from another

Printing an Object

- Default representation not very helpful
 - based on class name and hashCode
- Can override default by defining a `toString()` method
 - string returned from method used when printing object

```
public class Person {  
  
    // instance variables  
    private int age;  
    private String name;  
  
    // a constructor  
    public Person(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
  
    // ... setters and getters omitted for brevity  
  
    public String toString() {  
        return "Person(" + age + ", " + name + ")";  
    }  
}
```

Adding behaviour to the class Person

```
public class Person {  
    private int age;  
    private String name;  
  
    public Person(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
  
    public int getAge() {  
        return age;  
    }  
    public void setAge(int age) {  
        this.age = age;  
    }  
  
    public String getName() {  
        return name;  
    }  
    public void setName(String name) {  
        this.name = name;  
    }  
  
    public String toString() {  
        return "Person(" + age + ", " + name + ")";  
    }  
}
```

The PersonApp program

```
public class PersonApp {  
  
    public static void main(String[] args) {  
        Person p = new Person("John", 56);  
        System.out.println(p);  
  
        System.out.println(p.getName() + " is " + p.getAge());  
        p.setAge(57);  
        System.out.println(p.getName() + " is " + p.getAge());  
  
        System.out.println(p);  
    }  
}
```

Person [age=54, name=John]
John is 54
John is 55
Person [age=55, name=John]

The PersonApp expanded

```
public class PersonApp {  
  
    public static void main(String[] args) {  
  
        Person p1 = new Person("John", 56);  
        Person p2 = new Person("Denise", 53);  
        Person p3 = new Person("Phoebe", 23);  
        Person p4 = new Person("Adam", 21);  
        System.out.println(p1);  
        System.out.println(p2);  
        System.out.println(p3);  
        System.out.println(p4);  
    }  
}
```

Person [age=56, name=John]
Person [age=53, name=Denise]
Person [age=23, name=Phoebe]
Person [age=21, name=Adam]

Annotations

- Provide metadata about Java program elements
- Have the format
 - @<NameOfAnnotation>
- Can be applied to classes, constructors, methods, parameters, return values etc.
- Most common see is @Override
 - indicates method overrides another method through inheritance (but its optional)
 - used by the compiler to check that method signature is correct
 - IntelliJ adds annotation when autogenerate some methods

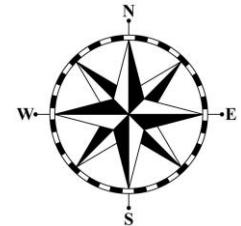
Packages

Toby Dussek

Informed Academy



Plan for Session



- Packages
- Class-Package Relationship
- Declaring Packages
- Packaging the Person class
- Role of the Classpath
- JAR Files
- Encapsulation and Packages
- Class modifier
- Constructor Modifier
- Variable Modifier
- Method Modifiers
- Package Summary

Packages

- Groupings of associated classes
- A class can belong to at most 1 package
- Can access a specific class
 - **import** java.util.ArrayList;
- Can access all classes in another package
 - **import** java.util.*;
 - Does not include sub-packages
- Packages can be named in a hierarchically style
 - e.g. java.util
- Always get **java.lang** – must import all other packages

Class-Package Relationship

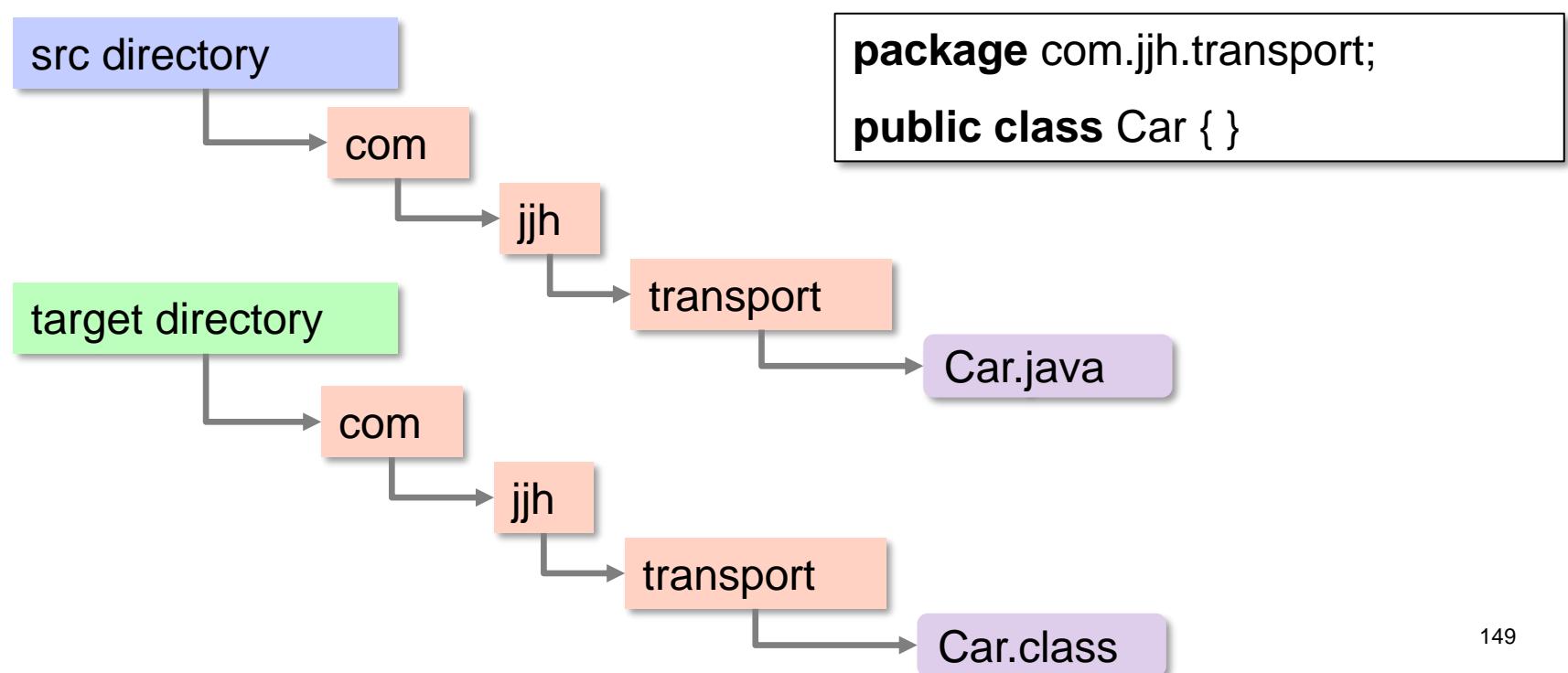
- All classes are always in a package
- If no package is explicitly defined, then class is added to the ***unnamed*** default package
- *Unnamed* package equates to
 - all classes in the current directory
 - that do not explicitly have a package declaration
 - classes in different directories – different unnamed packages
- As *unnamed* package has no name, cannot import it into another package
- In general do not use unnamed package!!

Declaring Packages

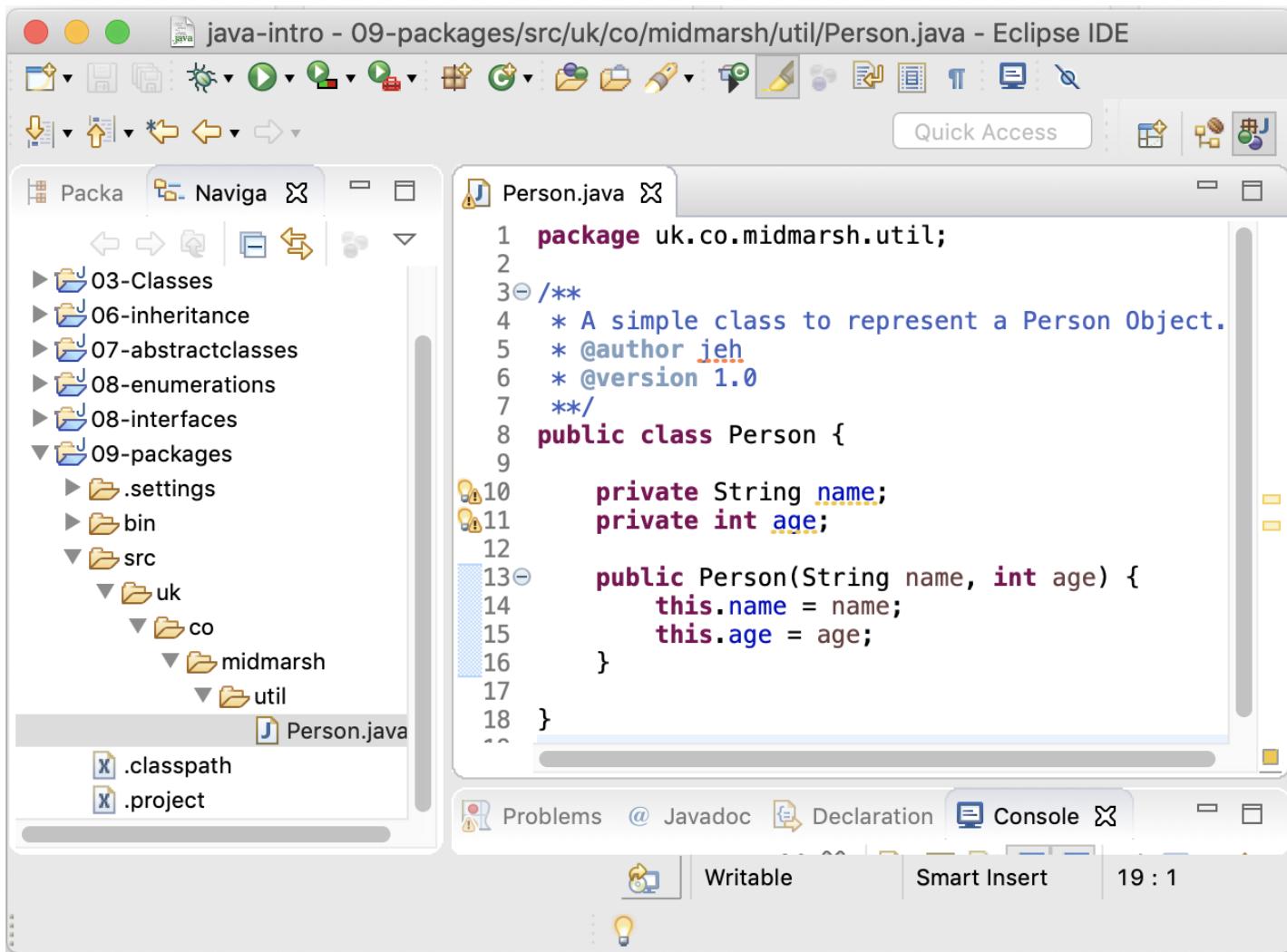
- Package declaration, if present, must be 1st statement
 - `package <package.name>;`
- Maximum one per file
- Package names are
 - lower case, separated by dots
 - are **not** hierarchical – although typically treated as such
 - typically represent domain names
 - `uk.co.midmarsh.util`
 - or libraries
 - `android.widget.Button`
 - have an associated directory structure

Package-Directory Relationship

- Relate to directory structure in which classes are held
 - compiler generates directory structure based on package
 - not enforced for source code (but common to follow)



Packaging the Person



The screenshot shows the Eclipse IDE interface with the following details:

- Title Bar:** java-intro - 09-packages/src/uk/co/midmarsh/util/Person.java - Eclipse IDE
- Toolbar:** Standard Eclipse toolbar with various icons for file operations, navigation, and code editing.
- Quick Access:** A button labeled "Quick Access" in the top right corner.
- Left Sidebar (Outline View):** Shows the project structure:
 - Packages: 03-Classes, 06-inheritance, 07-abstractclasses, 08-enumerations, 08-interfaces, 09-packages.
 - 09-packages folder contains .settings, bin, and src.
 - src folder contains uk, co, midmarsh, and util.
 - util folder contains Person.java.
- Central Editor:** The Person.java file is open in the editor. The code is as follows:

```
1 package uk.co.midmarsh.util;
2
3 /**
4  * A simple class to represent a Person Object.
5  * @author jeh
6  * @version 1.0
7 */
8 public class Person {
9
10    private String name;
11    private int age;
12
13    public Person(String name, int age) {
14        this.name = name;
15        this.age = age;
16    }
17
18 }
```
- Bottom Status Bar:** Shows tabs for Problems, Javadoc, Declaration, and Console, along with status indicators for Writable and Smart Insert, and the line number 19 : 1.

JAR Files

- Essentially a ZIP file with a manifest
- Created using jar command

```
jar cvf code.jar *.class
```

- Now contains all classes in current directory in compressed format
- Can then add to class path
- Or place into ext (extensions) directory

```
j2sdk1.4.2/jre/lib/ext
```

- Libraries usually provided as jar files
 - can then be packaged with an application

Role of the Classpath

- Classpath environment variable indicates where to look for packages

```
classpath= .;c:\classes;d:\temp;e:\project
```

- Can have jar files on classpath

```
classpath=.;c:\classes;d:\project\proj.jar
```

- System searches down class path looking for named class or package
- Can also provide on command line

```
java -classpath .;c:\classes com.midmarsh.main.Main
```

Encapsulation and Packages

- Packages are more than just a way of grouping classes
- Most encapsulation features in Java relate to packages
- Can make class, constructors, methods & variables visible to
 - the class
 - Just the package
 - Just the package and subclasses
 - Anywhere

Class Modifier

□ public

- class is visible everywhere
- just needs to be imported

```
package com.jjh.transport;  
public class Car { }
```

□ *no modifier*

- class is visible in current package
- does not need to be in own file
- cannot be imported into another package

```
package com.jjh.transport;  
class Car { }
```

Constructor Modifier

- ❑ public
 - constructor is visible everywhere
- ❑ no modifier
 - constructor visible in current package
- ❑ protected
 - visible in current package and subclasses anywhere
- ❑ private
 - only visible inside current class!

```
package com.jjh.transport;  
  
public class Car {  
    public Car() {}  
}
```

```
package com.jjh.transport;  
  
public class Car {  
    Car() {}  
}
```

```
package com.jjh.transport;  
  
public class Car {  
    protected Car() {}  
}
```

```
package com.jjh.transport;  
  
public class Car {  
    private Car() {}  
}
```

Variable Modifiers

- **public**
 - visible everywhere
 - class must also be public
- *no modifier* (package visibility)
 - visible only in current package
- **protected**
 - visible in current package and in subclasses in any package
- **private**
 - visible only to current class

```
package com.jjh.transport;  
  
public class Car {  
    private String brand;  
    public Car() {}  
}
```

Note protected is weaker than saying nothing at all!

Method Modifiers

- **public**
 - visible everywhere
 - class must also be public
- *no modifier* (package visibility)
 - visible only in current package
- **protected**
 - visible in current package and in subclasses in any package
- **private**
 - visible only to current class

```
package com.jjh.transport;  
  
public class Car {  
    private String brand;  
    public Car() { }  
    public void driver() { }  
}
```

Note protected is weaker than saying nothing at all!

Java Inheritance

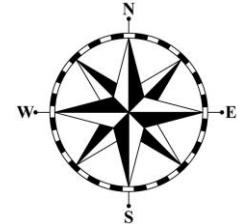


Toby Dussek

Informed Academy



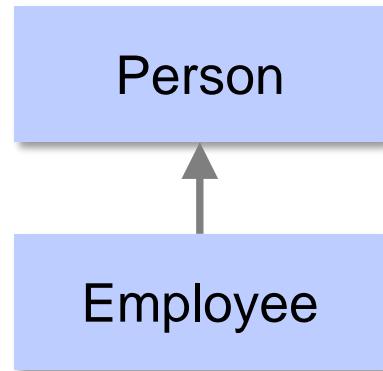
framework training
business value through education



Plan for Session

- Inheritance in Java
- Implementing Inheritance
- Rule for overriding methods
- Rules for Polymorphic variables
- Casting and Inheritance
- The super variable
- Constructors and Inheritance
- Use of Final keyword

Inheritance in Java

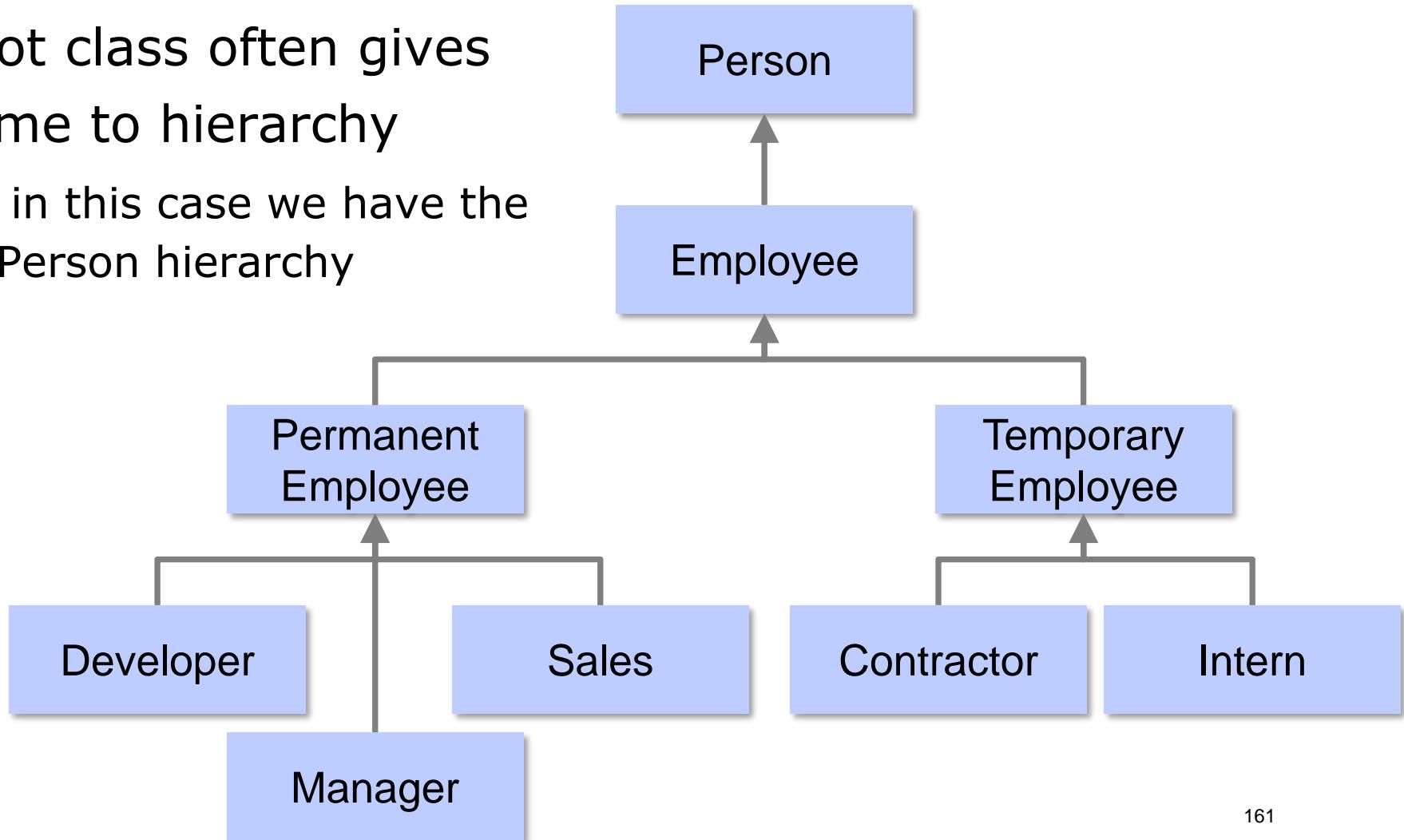


```
public class Employee extends Person {  
    // ...  
}
```

- Single inheritance
 - every class extends 1 other class
 - default is class `Object` if don't state a superclass

Implementing Inheritance

- Root class often gives name to hierarchy
 - in this case we have the Person hierarchy



Implementing Inheritance

```
public class Employee extends Person {  
    private int empNumber;  
}  
  
public class PermanentEmployee extends Employee {  
    private int numberOfDaysOff = 22;  
}  
  
public class Sales extends PermanentEmployee {  
    private double bonus = 200.0;  
}  
  
public class TemporaryEmployee extends Employee {  
    private String startDate;  
    private int periodOfContract = 6;  
}
```

Not just data in Inheritance

```
public class Employee extends Person {  
    public double calculatePay (int hours) {  
        return hourlyRate * hours;  
    }  
}  
  
public class Manager extends PermanentEmployee {  
    public void printPay() {  
        System.out.println(calculatePay());  
    }  
}  
  
public class Sales extends PermanentEmployee {  
    @Override // Optional annotation  
    public double calculatePay(int hours) {  
        return super.calculatePay(hours) + bonus;  
    }  
}
```

Rules for Overriding Methods

- Must have the same name
- Can't have less access than method being overridden
- Can't change return type
 - (although can have subclass of return type)
- Must have same parameter types
 - can have different argument names
- Must throw same type of exceptions as overridden method

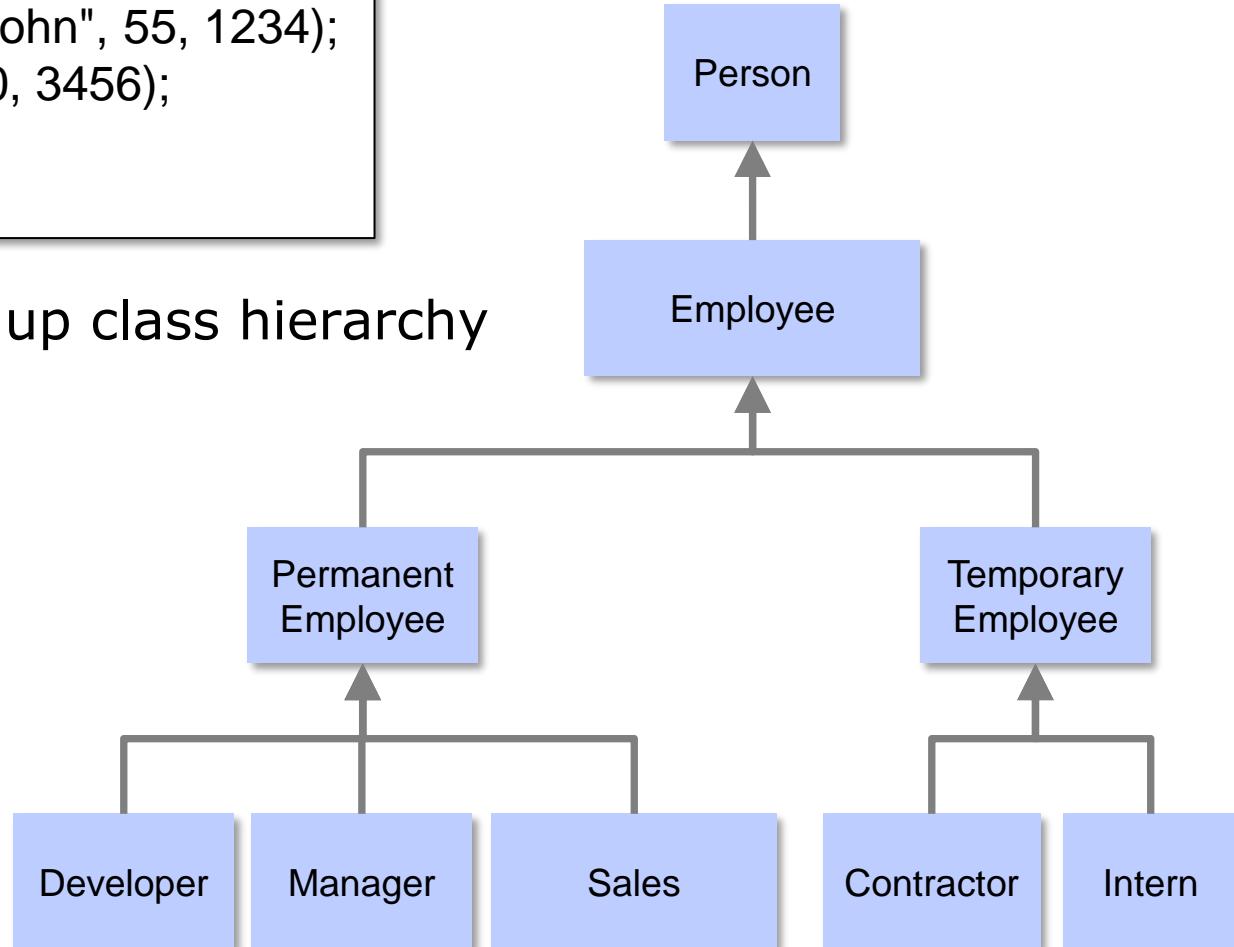
Rules for Polymorphic Variables

- A variable of type X
 - can be given objects of that type or
 - any descendant type
- Dynamic binding at runtime
- Types of variables are like “filters”
 - they say what methods can be executed on an object
 - even though an object may have other methods
 - not covered by the variable type and so not accessible.

Casting and Inheritance

```
Employee e = new Employee("John", 55, 1234);  
Sales s = new Sales("Adam", 20, 3456);  
e = s;  
System.out.println(e);
```

- Automatic casting back up class hierarchy



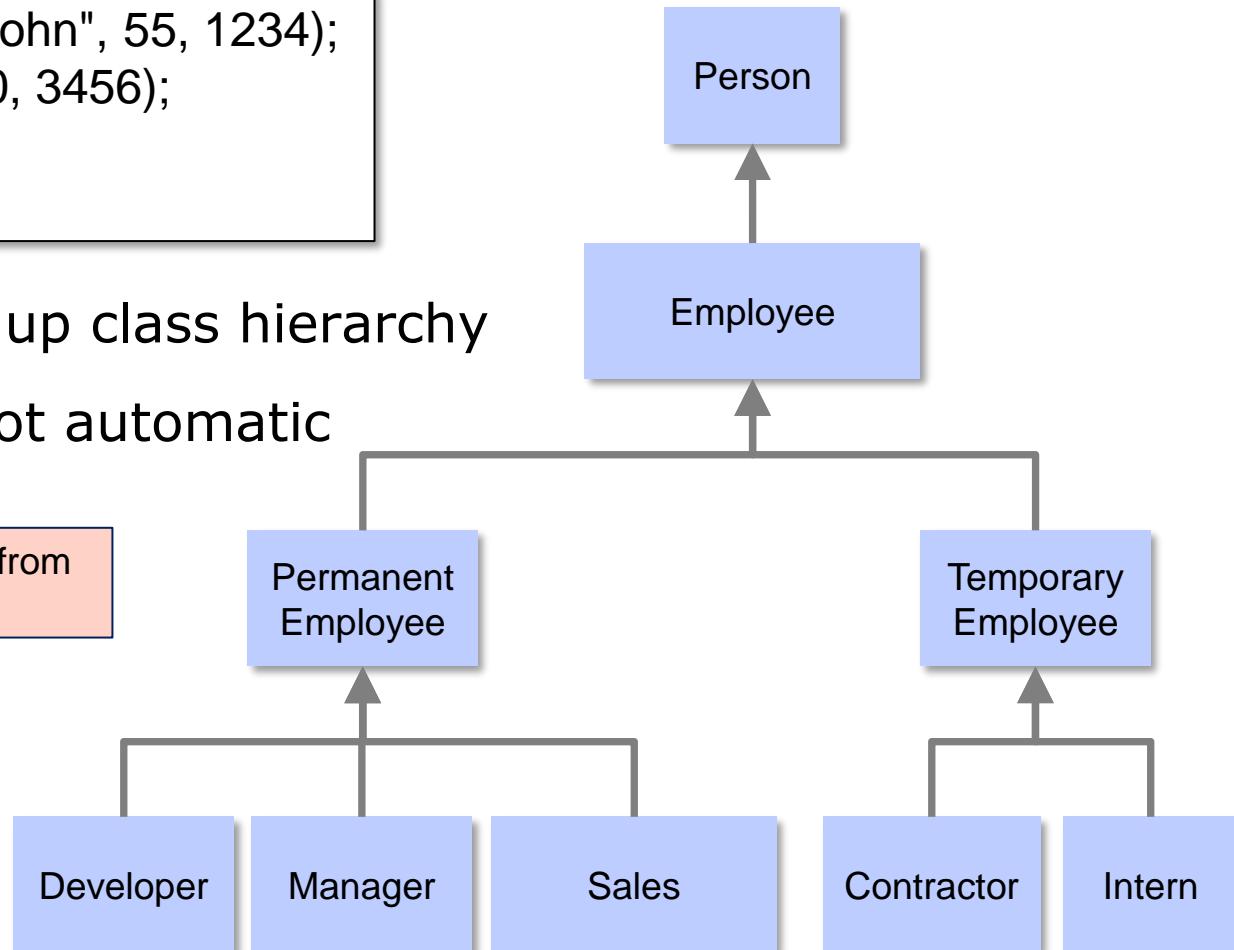
Casting and Inheritance

```
Employee e = new Employee("John", 55, 1234);  
Sales s = new Sales("Adam", 20, 3456);  
e = s;  
System.out.println(e);
```

- Automatic casting back up class hierarchy
- Casts down hierarchy not automatic

```
s = e;
```

Type mismatch: cannot convert from
Employee to Sales



Casting and Inheritance

```
Employee e = new Employee("John", 55, 1234);  
Sales s = new Sales("Adam", 20, 3456);  
e = s;  
System.out.println(e);
```

- Automatic casting back up class hierarchy
- Casts down hierarchy not automatic

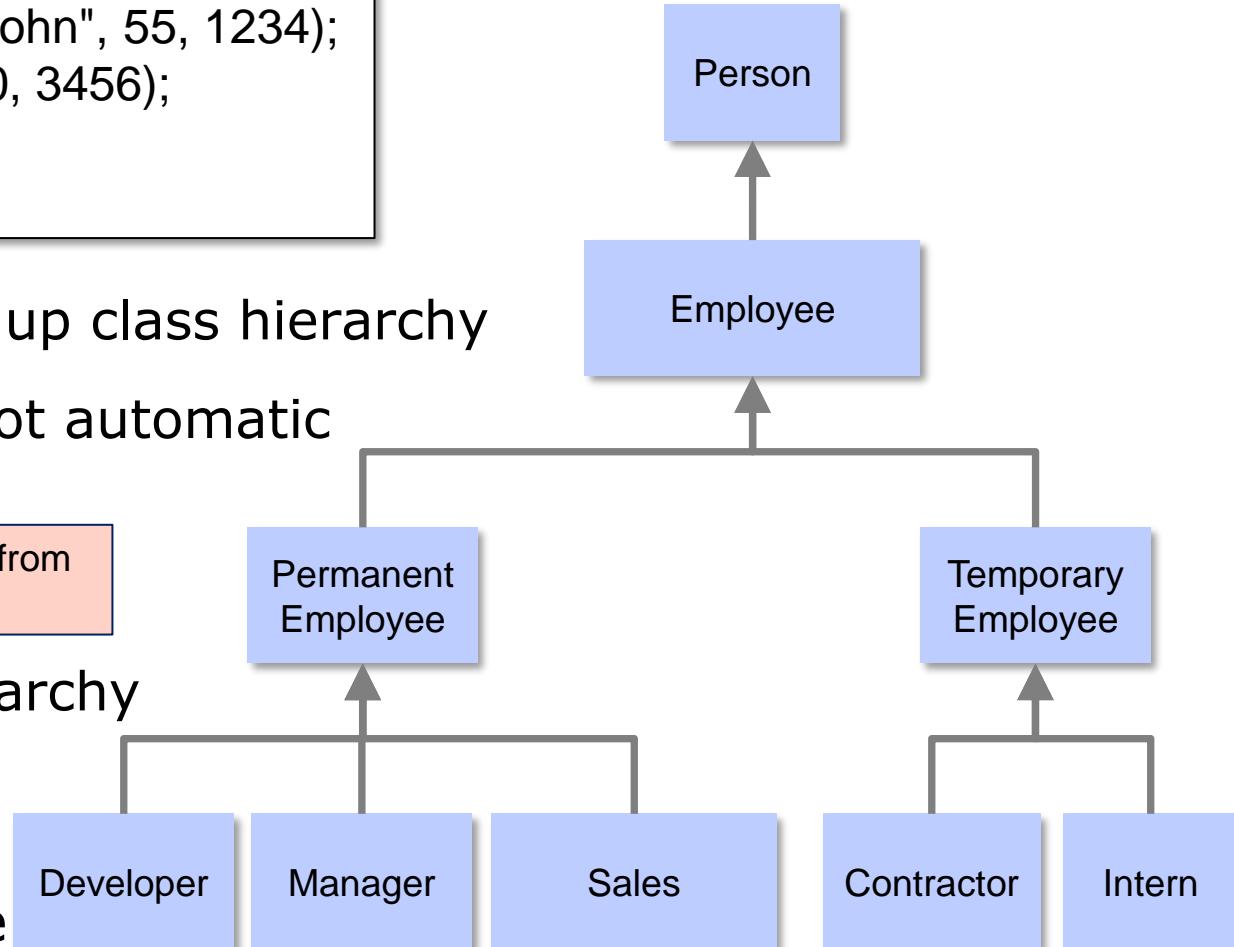
```
s = e;
```

Type mismatch: cannot convert from
Employee to Sales

- Explicit casts down hierarchy

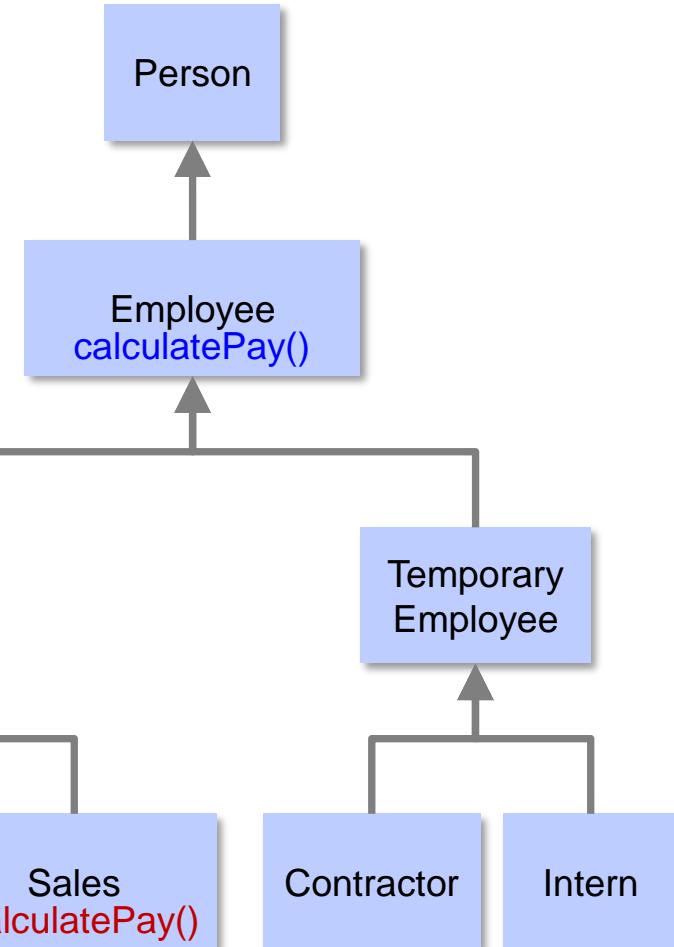
```
s = (Sales)e;  
System.out.println(s);
```

- May generate a runtime
error



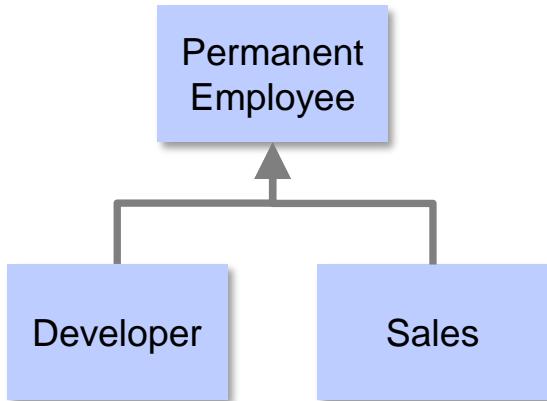
Inheritance relevant for methods

```
Employee e = new Employee("John", 55, 1234);
Sales s = new Sales("Adam", 20, 3456);
System.out.println(e.calculatePay(40));
System.out.println(s.calculatePay(40));
e = s;
System.out.println(e.calculatePay(40));
```



Which methods would be run?

Casting Issue



```
Employee e = new Employee("John", 55, 1234);  
Developer d = new Developer("Eloise", 25, 5432);  
Sales s = new Sales("Adam", 20, 3456);
```

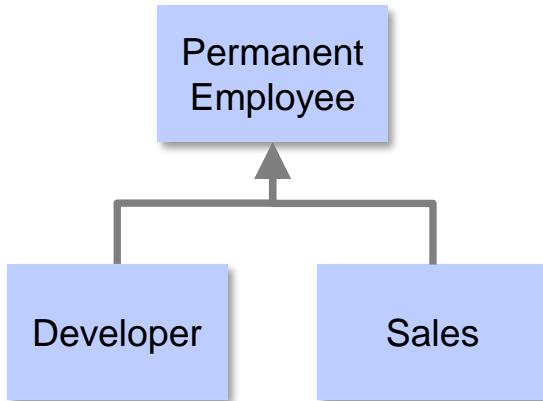
a `e = d;
e = s;`

b `d = s;
s = d;`

c `d = (Developer)e;`

- Which are valid?
- Which produce compile time errors?
- Which may produce runtime errors?

Casting Issue



```
Employee e = new Employee("John", 55, 1234);
Developer d = new Developer("Eloise", 25, 5432);
Sales s = new Sales("Adam", 20, 3456);
```

a `e = d;`
`e = s;`

b `d = s;`
`s = d;`

c `d = (Developer)e;`

- Which are valid?
- Which produce compile time errors?
- Which may produce runtime errors?

Casting and Basic Types

- For basic Types only safe casts performed automatically
 - safe from **int** to **long** (why?)
- Others you need to do manually:
 - **int** x = (**int**)(23.4 * 100);
 - change type of data unlike for reference types
- Can also specify literal types:
 - **long** l = 7L;
 - **float** f = 3.7f;
 - **double** d = 45.78d; // not required

The super variable

```
public class Person {  
    // ... omitted for brevity  
    public void sayHello() {  
        System.out.println("Hello");  
    }  
}
```

- Allows subclass to call super class version of a method

```
public class Employee extends Person {  
    // ...omitted for brevity  
    public void sayHello() {  
        System.out.println("Welcome");  
        super.sayHello();  
        System.out.println("Hi");  
    }  
}
```

```
Employee e = new Employee("John", 55, 1234);  
e.sayHello();
```

Allows for extension of behaviour

Welcome
Hello
Hi
173

Constructors and Inheritance

- Constructors are NOT inherited
- Super class Constructors are *always* called
 - will invoke zero parameter constructor if not explicitly called
 - will result in error if zero param constructor does not exist
- Can make one constructor explicitly call another
 - **this(<params>)**
 - Looks in current class – seen this already
 - **super(<params>)**
 - looks in parent class
- Must be first line of constructor
- One constructor always calls another
 - but only one either implicitly or explicitly

Invoking superclass constructors

- Using `super` to call `Person` class constructor

```
public class Employee extends Person {  
    private int id;  
    private int empNumber;  
    private double hourlyRate = 24.00;  
  
    public Employee(String n, int a, int id) {  
        super(n, a);  
        this.setId(id);  
    }  
  
    // ...  
}
```

Final Keyword

- Indicates that a value, method or class
 - cannot be modified lower in the class hierarchy
- Final Classes (cannot be extended)
 - e.g. String
 - **public final class** String { ... }
- Final Methods (cannot be overridden)
 - **public final void** printIt()...
- Final variables
 - in effect constants (also class variables)
 - **public final int** max = 100;
- Parameters and local variables
 - constants for that method call

```
public final class Student {  
    private String name;  
    private int age;  
    private String subject;  
    private int result;  
    public final int year = 2020;
```

```
public Student(String name, int age, String subject, int result) {  
    this.name = name;  
    this.age = age;  
    this.subject = subject;  
}  
public final void prettyPrint() {  
    System.out.println("Student(" + name + ", age: " + age +  
        " studying" + subject + " with " + result);  
}  
public void setAge(final int newAge) {  
    this.age = newAge;  
}  
public String calculateStatus() {  
    final int fail = 100;  
    if (result < 50) {  
        return "Fail";  
    }  
    return "Pass";  
}
```

Final class with final year, final method and final parameters and local constant

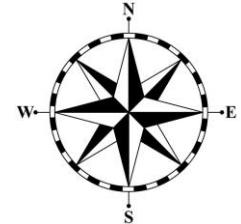
Abstract Classes and Class Side Behaviour



Toby Dussek

Informed Academy





Plan for Session

- Abstract Classes
- Abstract Classes in Java
- Defining an Abstract Class
- Extending an Abstract Class
- Using Concrete Subclasses
- Class Side Information
- Class Side Data
- Class Side Behaviour

Abstract Classes

- A class with some aspect that is abstract
 - you are not allowed to make instances from it
 - can only make instances from concrete classes
 - may contain concrete methods and properties
 - may also contain abstract methods and properties
- Used when
 - wish to specify data or behaviour common to a set of classes, but insufficient for a single instance,
 - you wish to force subclasses to provide specific behaviour
- Good OO practice

Abstract Classes in Java

□ Abstract Classes

- indicated by the keyword `abstract`
- cannot be instantiated themselves
- but can be extended by subclasses
- have Zero or more abstract methods
- have Zero or more concrete methods

□ Many built-in abstract classes including

- data structures / collections
- streams

Defining Abstract Classes

- The keyword `abstract` used to
 - define an abstract class as a whole
 - to define an abstract method
 - a method with a signature but no body
 - if method is abstract then class *must* be marked as abstract
- Subclasses can extend abstract class
 - must implement all abstract methods
 - if they don't then subclass must be marked as abstract

Defining Abstract Classes

□ Abstract class with abstract method

```
public abstract class Conveyance {  
  
    protected double fuel = 5.0;  
    private boolean running;  
  
    public void startup() {  
        this.running = true;  
        this.consumeFuel();  
        while (this.fuel > 0) {  
            this.consumeFuel();  
        }  
        this.running = false;  
    }  
  
    public abstract void consumeFuel();  
}
```

*Subclasses must implement
the consumeFuel ()
method or be abstract
themselves*

Extending an Abstract Class

- Can create subclass from abstract class
 - no special syntax needed
 - can use optional `@Override` annotation if desired
 - not a requirement

```
public class Car extends Conveyance {  
  
    @Override  
    public void consumeFuel() {  
        fuel = fuel - 0.5;  
        System.out.println("consuming, ");  
    }  
}
```

Extending an Abstract Class

- Subclass must be marked as abstract
 - must be marked as abstract if it fails to implement one or more abstract methods
 - Generates a compile time error if class is not marked as abstract

```
public class CommercialVehicle extends Conveyance {  
}  
The type CommercialVehicle must implement the inherited abstract method  
Conveyance.consumeFuel()
```

Using Concrete subclasses

- Concrete subclass of abstract class
 - used in normal manner
 - can inherit behaviour from abstract class
 - can be treated as class or superclass

```
public class Test {  
  
    public static void main(String[] args) {  
        Conveyance c = new Car();  
        c.startup();  
    }  
}
```

consuming,
consuming,

Class Side Information

- Classes aren't just templates!
- Can hold data
 - Class / Static variables and *constants*
- Can hold methods
 - Class / Static methods
- The static keyword used for both
- Can be used for housekeeping
- Can be useful for generic functions
- Static methods are inherited

Class Side Data

- Keep track of number of instances created
 - instances variable part of class (not objects)
 - each time a new instance is created, instances is incremented

```
public class Manager {  
  
    public static int instances;  
  
    public Manager() {  
        instances++;  
    }  
}
```

```
public class ManagerApp {  
    public static void main(String[] args) {  
        Manager m1 = new Manager();  
        Manager m2 = new Manager();  
        Manager m3 = new Manager();  
        System.out.println(Manager.instances);  
    }  
}
```

Class Side Behaviour

- Implementing the singleton pattern
 - can't create a new instance of Session outside of class
 - can only obtain instance via *static getInstance()*
 - single instance held in class side (private) singleton variable

```
public class Session {  
    private static Session singleton;  
  
    public static Session getInstance() {  
        if (singleton == null) {  
            singleton = new Session();  
        }  
        return singleton;  
    }  
  
    private Session() {} // no public instantiation  
}
```

Class Side Behaviour

- `Session.getInstance()` always returns the same instance of the Session

```
public class SessionApp {  
    public static void main(String[] args) {  
        Session s1 = Session.getInstance();  
        System.out.println(s1);  
        Session s2 = Session.getInstance();  
        System.out.println(s2);  
        Session s3 = Session.getInstance();  
        System.out.println(s3);  
    }  
}
```

Session@4b1210ee
Session@4b1210ee
Session@4b1210ee

- Note `main` method is a *static* method

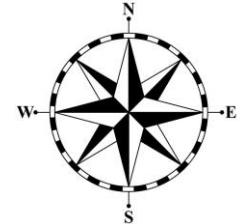
Java Interfaces & Enumerations



Toby Dussek

Informed Academy





Plan for Session

- What is an Interface in Java?
- Basic Interface Definitions
- Implementing an Interface
- Interfaces and Types
- Using an Interface in a Contract
- Inheritance by Interfaces
- Classes and Multiple Interfaces
- Default Interface Methods
- Static Interface Methods
- Enumeration Support
- Implementing Enumerated Types

What is an Interface in Java?

- ... a skeleton which specifies the protocol that a class *must* provide if it implements that interface...
- Basic format of an Interface:

```
access-modifier interface interface-name {  
    method signatures ...  
}
```

- Acts as a specification for code
- Allows for pluggability in code
- Classes *implement* interfaces

Basic Interface Definitions

□ An example interface

```
public interface Organizer {  
  
    // Following methods are public abstract  
    void add(String appointment, String date);  
  
    String get(String date);  
  
    public boolean remove(String date);  
  
}
```

- if public must be defined in own .java file with same name as interface

Implementing an Interface

- Class can *implement* one or more interfaces
 - but must implement all methods in the interface
 - otherwise class must be marked as abstract
 - can *implement* methods via inheritance

```
public class Calendar implements Organizer {  
  
    public void add(String appointment, String date) {  
        System.out.println(appointment + " - " + date);  
    }  
  
    public String get(String date) {  
        return null;  
    }  
  
    public boolean remove(String date) {  
        return false;  
    }  
}
```

Interfaces and Types

- Instances of `Calendar` can be treated as:
 - `Calendar` or `Organizer`
 - all `Calendar` objects are also types of `Organizer`
 - but not all `Organizers` may be types of `Calendar`

```
public class CalendarApp {  
    public static void main(String[] args) {  
        Calendar cal = new Calendar();  
        cal.add("Dentist", "Monday");  
        Organizer org = cal;  
        org.add("Garage", "Tuesday");  
    }  
}
```

Using an Interface as a Contract

- Can use interface name to specify type of object:

```
public class Diary {  
    // ....  
    public void add (Organizer temp) {  
        // ....  
    }  
}
```

- Object passed to `add` is guaranteed to understand whole of `Organizer` protocol

Inheritance by Interfaces

- Interfaces can inherit from interfaces
- Interfaces inherit specifications
- Use the **extends** keyword to indicate inheritance:

```
public interface Records extends Workers, Employers, Cloneable {
```

```
// ...
```

```
}
```

- Notice interface can extend more than one interface
 - result is the union of all declarations

Classes and Multiple Interfaces

- Classes can implement multiple interfaces

```
public class Application implements Cloneable,  
                                Organizer,  
                                Printer,  
                                Speaker {  
  
    public void saySomething() {}  
    public void prettyPrint() {}  
    public void add(String appointment, String date) {}  
  
    public String get(String date) {  
        return null;  
    }  
  
    public boolean remove(String date) {  
        return false;  
    }  
}
```

Default Interface Methods



- Java8 Interfaces extend concept of an interface
- Interfaces can contain default methods
 - methods marked by keyword **default** with a method body
 - as well as abstract methods

```
public interface Speaker {  
  
    public void saySomething();  
  
    default public void sayHello() {  
        System.out.println("Speaker - Hello World");  
    }  
  
}
```

- Interfaces more like a Mixin



Default Interface Methods

- Class can implement interface
 - and inherit default method implementation

```
public class Person implements Speaker {  
    @Override  
    public void saySomething() {  
        System.out.println("Person - Howdy");  
    }  
}
```

- or override if required - @Override is optional
- Both methods available on object

```
public class PersonApp1 {  
    public static void main(String[] args) {  
        Person p = new Person();  
        p.sayHello();  
        p.saySomething();  
    }  
}
```

Speaker - Hello World
Person - Howdy

Default Interface Methods



- Multiple interfaces can have same default methods

```
public interface Translator {  
    default public void sayHello() {  
        System.out.println("Actor - Bonjour");  
    }  
}
```

```
public interface Speaker {  
    public void saySomething();  
  
    default public void sayHello() {  
        System.out.println("Speaker - Hello World");  
    }  
}
```

```
public class Employee implements Speaker, Translator{  
    public void saySomething() {  
        System.out.println("Employee- Say Something");  
    }  
    public void sayHello() {  
        System.out.println("Employee - Hello");  
    }  
}
```

Employee - Hello
Employee- Say Something

Default Interface Methods



- Can call interface method if required

```
public interface Translator {  
    default public void sayHello() {  
        System.out.println("Actor - Bonjour");  
    }  
}  
  
public interface Speaker {  
    public void saySomething();  
    default public void sayHello() {  
        System.out.println("Speaker - Hello World");  
    }  
}  
  
public class Employee implements Speaker, Translator{  
    public void saySomething() {  
        System.out.println("Employee- Say Something");  
    }  
    public void sayHello() {  
        Speaker.super.sayHello();  
    }  
}
```

Speaker - Hello World
Employee- Say Something

Static Interface Methods



- Interfaces can also have static methods
 - part of the interface not the instance

```
public interface Printer {  
  
    void prettyPrint();  
  
    static void printMe() {  
        System.out.println("Printer - printMe");  
    }  
}
```

```
public class ShoppingBasket implements Printer {  
  
    public void prettyPrint() {  
        System.out.println("ShoppingBasket - prettyPrint");  
    }  
}
```

Static Interface Methods



- Can call static method on interface
 - but cannot call it on implementing class
 - nor can it be invoked on object of class

```
public class ShoppingBasketApp {  
  
    public static void main(String[] args) {  
        // Can call  
        Printer.printMe();  
        // Compile error  
        // ShoppingBasket.printMe()  
        ShoppingBasket basket = new ShoppingBasket();  
        basket.prettyPrint();  
        // Compile error  
        // basket.printMe();  
    }  
}
```



Java 9 Further Extensions

- Provides for private static / non-static methods

```
public interface Java8StyleInterface {  
    default void logWarning(String msg) {  
        log("Warning", msg);  
    }  
    default void logError(String msg) {  
        log("Error", msg);  
    }  
    private void log(String level, String msg) {  
        System.out.println(level + ": " + msg);  
    }  
    default void checkStatus(String status) {  
        if (isNull(status)) {  
            System.out.println("Unknown status");  
        }  
    }  
    private static boolean isNull(String str) {  
        return str == null ? true : "".equals(str) ? true : false;  
    }  
}
```

Enumeration support

- Introduced in Java 5
- Provides for common Enumerated type concept
- Introduced enum keyword
- Defined in a normal Java source file
- Each value of the enum has two principal attributes
 - name
 - Ordinal (Zero based)
 - accessible through name() and ordinal() methods
- Default implements provided for
 - equals(), hashCode(), toString(), compareTo()

Implementing Enumerated types

- Language support for the enumeration types
 - comma separated list of values

```
public enum Direction {  
    NORTH, SOUTH, EAST, WEST  
}
```

- Can use as a type to declare variables
- Can use to provide values, perform tests etc.

```
public class DirectionApp {  
    public static void main(String[] args) {  
        Direction d = Direction.NORTH;  
        System.out.println(d);  
        if (d == Direction.NORTH) {  
            System.out.println("We are heading North");  
        }  
    }  
}
```

NORTH
We are heading North 208

Additional Enumeration Options

- May contain member fields and methods
- Including one or more constructors
 - constructors may only be private (or default package) visible

```
public enum CompassDirections {  
  
    NORTH(0), SOUTH(180), EAST(90), WEST(270);  
    private final int bearing;  
  
    private CompassDirections(int bearing) {  
        this.bearing = bearing;  
    }  
  
    public int getBearing() {  
        return this.bearing;  
    }  
}
```

Enumeration Values

- Enumerated values can be dynamically discovered:
- May also be used in switch statements:

```
public class CompassDirectionsApp {  
    public static void main(String[] args) {  
        CompassDirections[] directions = CompassDirections.values();  
        System.out.println(directions.length);  
  
        CompassDirections d = CompassDirections.WEST;  
        switch (d) {  
            case WEST:  
                System.out.println("Heading " + d.getBearing());  
                break;  
            default:  
                System.out.println("Unknown heading");  
        }  
    }  
}
```

Class Design

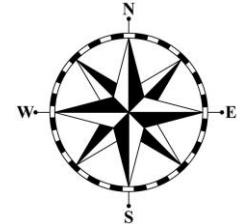


Toby Dussek

Informed Academy



framework training
business value through education



Plan for Session

- Naming Classes
- SOLID Principles
- Design Patterns
 - Singleton, Adapter, Strategy, Observer, Factory, Façade
 - Combining Design Patterns
 - Model-View-Controller
- Anti Patterns
- Code Smells

Naming Classes

- Provide a large amount of meaning / semantics
 - Makes it easier to understand not only the class but also where its used
 - Easier to discuss with others / on board newcomers
- Some rules of thumb
 - be descriptive in class names
 - e.g. BookCatalog, BookRepository, BookDataLoader
 - use short hand only when meaningful
 - e.g. BookDAO, BookDTO are only meaningful if DAO and DTO are understood
 - Remove superfluous words
 - e.g. SpecialBookCatalog – why is this special; is this really a new class?

SOLID: Class Design

- **S** – Single Responsibility Principle
- **O** – Open-Closed Principle
- **L** – Liskov Substitution Principle
- **I** – Interface Segregation Principle
- **D** – Dependency Inversion Principle

Single Responsibility Principle

- A class should only have a single responsibility
- It should only do one thing
 - but do that one thing well
- Easy to get wrong
 - as classes hold both data and behaviour
 - easy to merge behaviours
- If this happens refactor
 - so that one class delegates to another
- Problems naming a class may indicate multiple responsibilities

Single Responsibility Principle

- To see if you are adhering to the SRP
- Check that
 - the description of the class is short and clear? If not, is this a reflection on the class?
 - if the description is short and clear, do the class and attributes make sense within the context of the description?
 - look at how and where the attributes of the class are used
 - is their use in line with the class description?
 - and do the same for methods; are they appropriate in the context of the class

Open-Closed Principle

- Classes should be
 - open for extension; but closed for modification
- That is, if required
 - should be possible to extend behaviour
 - without need to change class's source code
- This could be done via
 - inheritance by subclasses
 - or delegation to functions or instances of other classes
- Using delegation
 - can register a function or object to perform some required behaviour

Liskov Substitution Principle

- Objects in a program
 - should be replaceable with instances of their subtypes
 - without altering the correctness of that program
- Implies design by contract
 - key idea behind polymorphism
 - also behind the concept of a *Interface* in Java
 - can be supported by Abstract Classes
- For example
 - given an online product ordering system
 - it should not matter what type of product
 - you should still be able to order them, and pay for them etc.

Liskov Substitution Principle and Classes

- Classes encapsulates Data and Functions
 - *interface* offered by class supported by subclasses
- Defines a type of object from the problem domain
 - may have specializations of that type of object
 - but still have (at least) same interface
- Classes hide implementation
 - state changes are achieved through public functions
 - internal workings and state are hidden from user
 - implementation can be changed without affecting user
 - minimal coupling / high cohesion

Interface Segregation Principle

- Many client-specific interfaces
 - are better than one general-purpose interface
- Developers find it easier to work with specific interfaces
 - that only address their direct concerns
 - large interfaces with many optional methods can be confusing
- Typically means
 - large interfaces should be split into smaller, specific purpose ones
- In Java this may mean
 - Multiple Interfaces being described for different purposes
 - Use of Abstract Classes to provide a set of required behaviours

Dependency Inversion Principle

- Depend on abstractions not on concrete types
- Principle states
 - High-level classes should not depend on low-level classes
 - Abstractions should not depend on details
- Helps to decouple implementations
- In Java depend on an Interface
 - allows flexibility in which classes implement the interface
- Or use Abstract classes to create an abstract type
 - that will require one or more behaviours to be provided

Designing Interacting Classes

- Is complex
 - to get right
 - to be clear
 - to determine how and when to use what styles / approaches
- Design Patterns to the rescue!





What are Design Patterns?

- Repeating abstract design solutions to a common problem
 - “Oh I’ve seen this problem before and I solved it by...”
- As at the design level they
 - generally require some modification or need to be applied
- Last time I put up a shelf I did this, but this time it’s a corner shelf so I now need to also

“captures expertise describing an architectural design to a recurring design problem in a particular context” – Buschmann et all

- includes knowledge of applicability of a pattern, trade offs, and any consequences of the solution

Pattern Classifications

□ GoF classifications

- Creational Patterns
- Structural Patterns
- Behavioural Patterns

E. Gamma, R. Helm, R. Johnson and J. Vlissades, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995, 0-201-63361-2



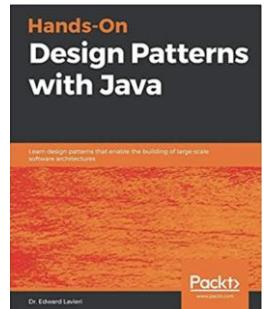
□ Architectural Patterns

- Used to describe higher-level structures

- e.g. Model-View-Controller
- Also Layers, Blackboard, micro-kernel
- Microservice design patterns
 - <https://microservices.io/patterns/microservices.html>
 - <https://dzone.com/articles/design-patterns-for-microservices>

□ Java coding patterns

- lead to Idiomatic Java



GoF Pattern Classifications

	Creational	Structural	Behavioural
<i>Class-based, static, inheritance</i>	Factory Method	Class Adapter	Interpreter Template Method
<i>Dynamic, run-time, association</i>	Abstract Factory Builder Prototype Singleton	Object Adapter Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

From “Design Patterns: Elements of Reusable Object-Oriented Software” by Gamma et al.

The Singleton Pattern

□ Intent

- Ensure a class only ever has one instance and provide a global point of access

□ Motivation

- In some situations there should only be one instance of a class to ensure that all objects access the same object. For example, although there may be many printers, there should be a single print spooler.

□ Solution

- Make the class responsible for creating one instance and keep track of its sole instance. 'User' code must then access class for instance.

Pattern Example: Singleton

- Only allow single instance of a class

```
public class Singleton {  
  
    private static Singleton instance = null;  
  
    public static synchronized Singleton getInstance() {  
        if (instance==null) {  
            instance = new Singleton();  
        }  
        return instance;  
    }  
  
    private Singleton() {}  
}
```

```
Singleton s = Singelton.getInstance()
```

Pattern Example: Adapter



□ Intent

- To convert the interface of one type into the interface that the client expects. Thus the adapter pattern allows types to work together that could not otherwise inter operate.

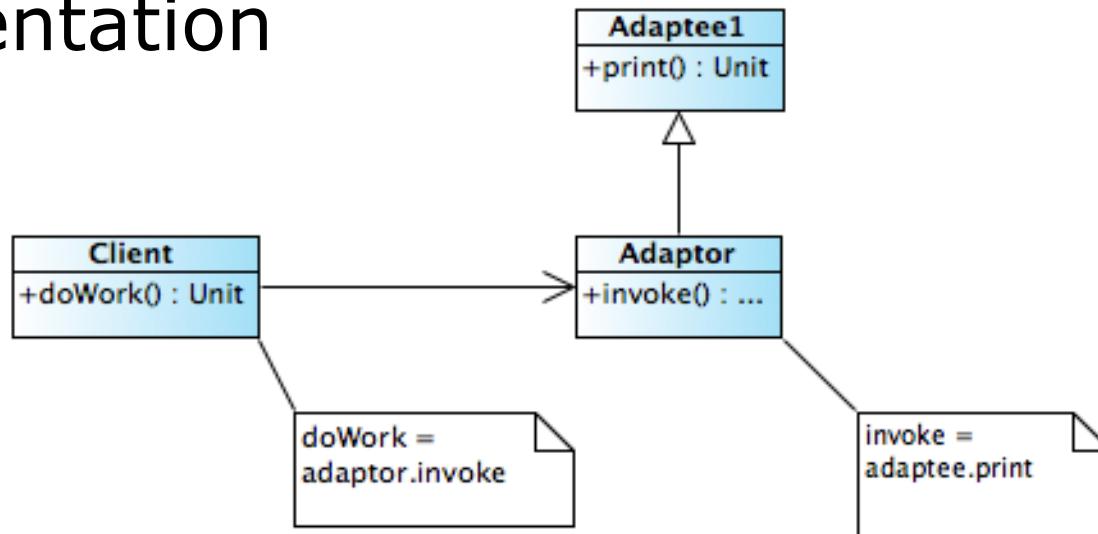
□ Motivation

- To link together two types in which the operations are semantically the same but are not named the same.
- To link together two types in which are functionally the same but where they are not related in the type system.
- To link together two types in which the data formats required are transformable but not directly compatible
- Some combination of the above.

Pattern Example: Adapter



- This approach exploits inheritance to allow an Adapter to extend the Adaptee and to provide the link between the Client's expectations and the Adaptees implementation



- Alternatively use delegation

Pattern Example: Adapter



□ Pros

- Client and Adaptee classes remain independent of each other
- Adapter can dynamically determine which of the Adaptees operations to invoke allowing for both flexibility

□ Cons

- introduces an additional indirection into the program
 - and thus an additional method call(s)
- may contribute to the complexity in understanding the implementation

Pattern Example: Strategy

□ Intent

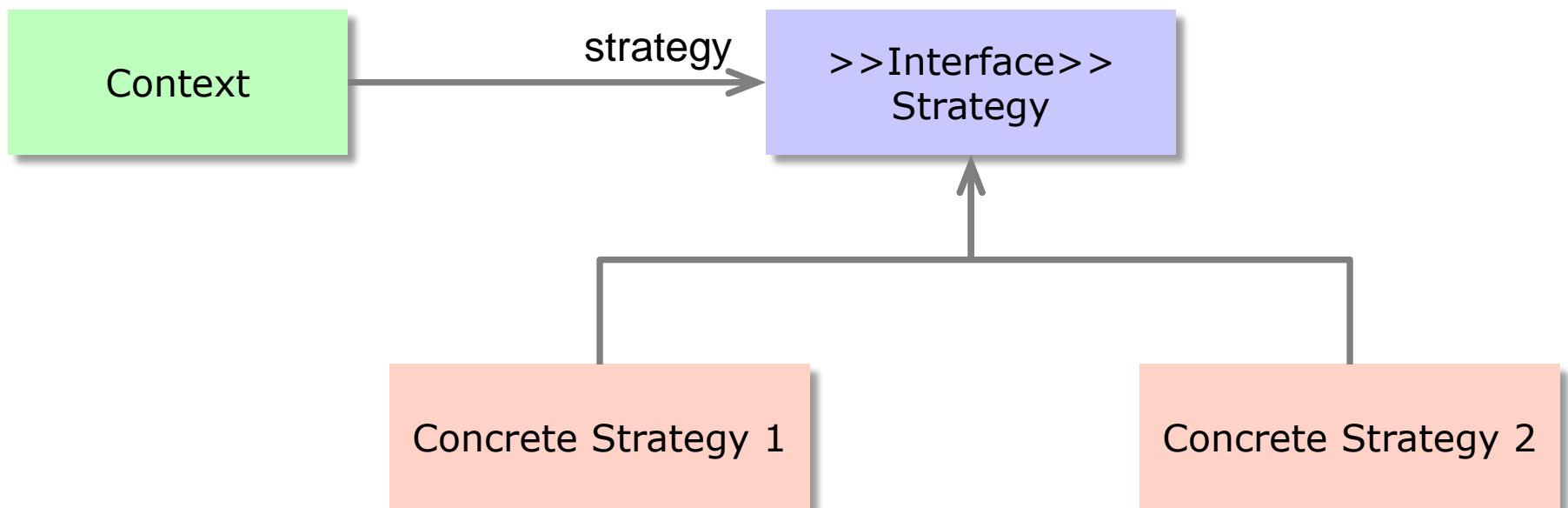
- Allows algorithmic implementation of some behaviour to be selected as required (and to be changeable over time).

□ Motivation

- An application must be able to change the implementation of an operation at runtime possibly based on some selection criteria
- The differences in implementation (algorithm) can be encapsulated into the strategy and there is a consistent approach to accessing these different implementations
- Clients do not need to know anything about how the operations are implemented
- Switching between different instances of different classes would loose identity integrity

Pattern Example: Strategy

- Allow inter-changeable implementations of algorithms
 - i.e. can select algorithm to apply at runtime
 - works very well with FP features in Java, Scala, Python, JavaScript



Pattern Example: Strategy

□ Pros

- allows behaviour of clients to be determined dynamically, at runtime, potentially on a per invocation basis
- Client objects can be simplified, as they do not need to be responsible for the selection of the appropriate behaviour

□ Cons

- It may not be clear at compile time which strategy will be employed and this may make debugging and maintenance tasks harder

Pattern Example: Observer

□ Intent

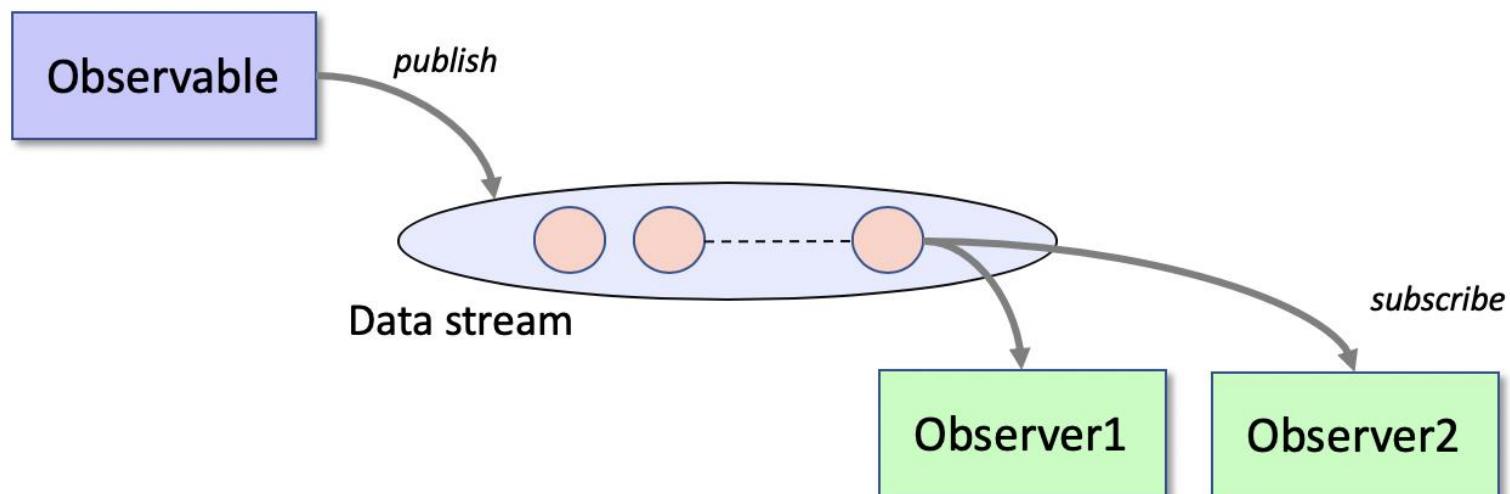
- To manage a one to many relationship between an object and those objects interested in the state and in particular state changes of that object.

□ Motivation

- Two, otherwise independent types, are related by the changes in the state of one (which need to be notified to the other).
- Where a change in the state of one object needs to be notified to zero or more other objects and where the number or actual objects may change or are not known until runtime.

Pattern Example: Observer

- When state change happens in observable
 - all observers are notified



- Observers react to changes in Observables state
- Core of RxJava reactive programming library

Pattern Example: Observer

□ Pros

- Loose Coupling. The pattern allows an object to deliver notifications of changes to other objects without the objects being directly aware of each other's types.
- Support for broadcast mechanisms. a single change notification can be broadcast to multiple Observers. And actual broadcast mechanism hidden

□ Cons

- Unpredictable notification times. Notification of changes to Observers can be a time consuming process
- Can have cyclic dependencies

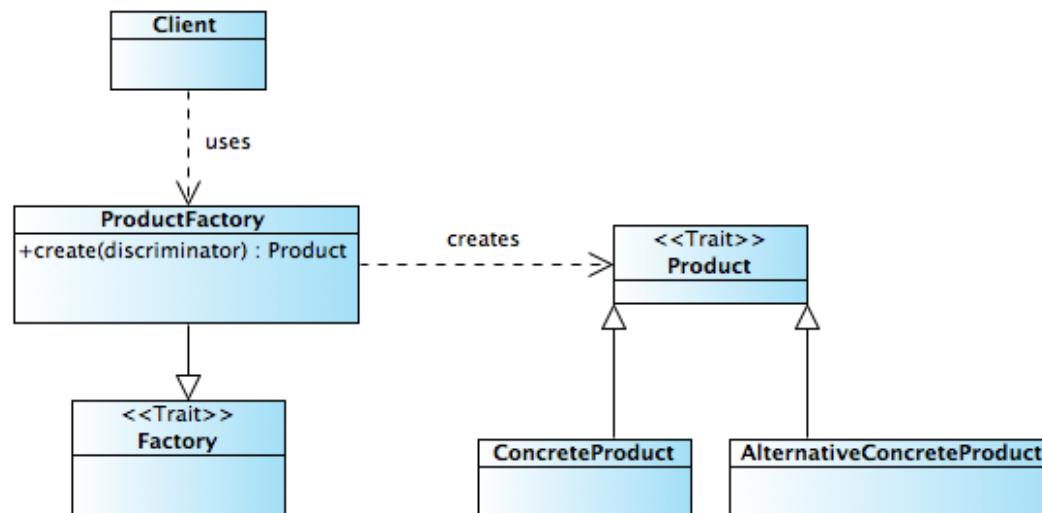
Pattern Example: Factory

□ Intent

- To hide the instantiation of a specific type from a client thereby allowing flexibility in that instantiation

□ Have a Factory class to create objects

- may use a factory method on product type



Pattern Example: Factory

□ Pros

- client and creation process separated / independent
- actual type being instantiated is hidden from the client
 - can therefore change without the client code needing to be altered.
- The set of product types being instantiated may change dynamically and independently of the client

□ Cons

- separation of the instantiation process from the client code
 - may therefore not be clear until run time what the actual product type instantiated

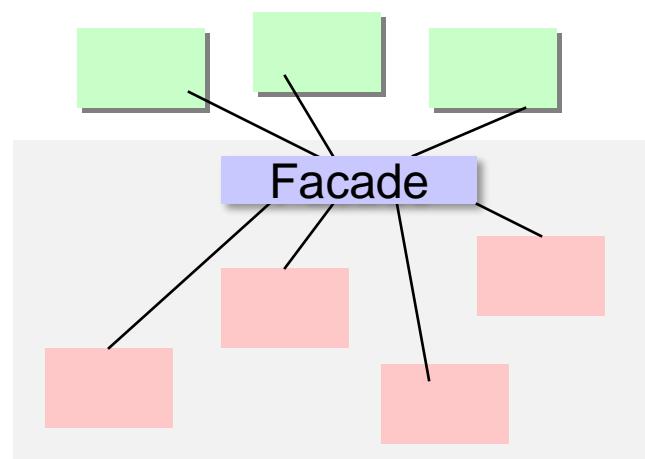
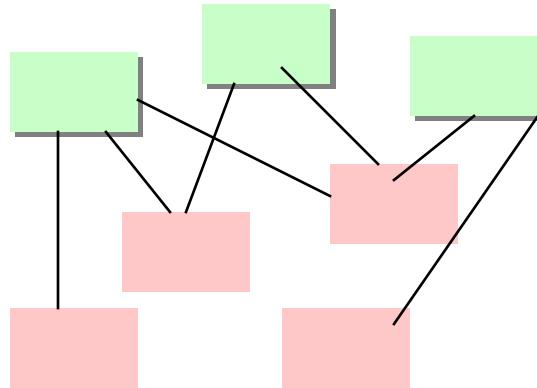
Pattern Example: Façade

□ Intent

- Provide a unified interface to a set of objects. Defines a higher level interface.

□ Motivation

- Easier to maintain one link than many, reduces dependencies between classes, helps reduce complexity.



Pattern Example: Façade

□ Pros

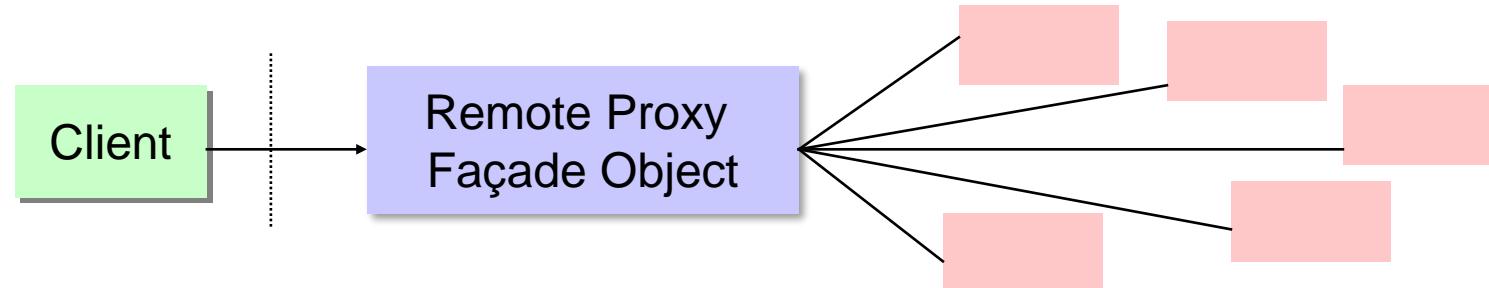
- Clients of the Façade do not need to know anything about the objects behind the façade
- Single dependency between the client and Façade
- Changes can be made to the objects behind the Façade with little or no impact on the client

□ Cons

- Façade must be able to deal with core interactions
- Client may need to *get behind* the Façade for edge case or exception scenarios.

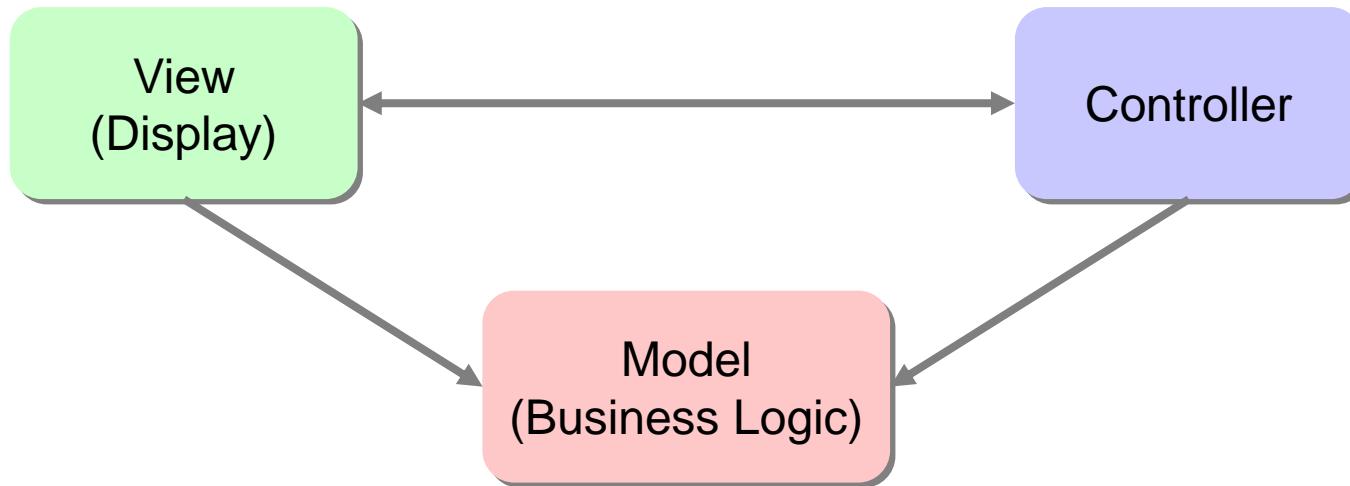
Combining Design Patterns

- Common to identify the need for more than 1 pattern
- Patterns may be combined to solve a particular problem
- For example
 - May combine Façade and Proxy in an client-server application



The Model-View-Controller Architecture

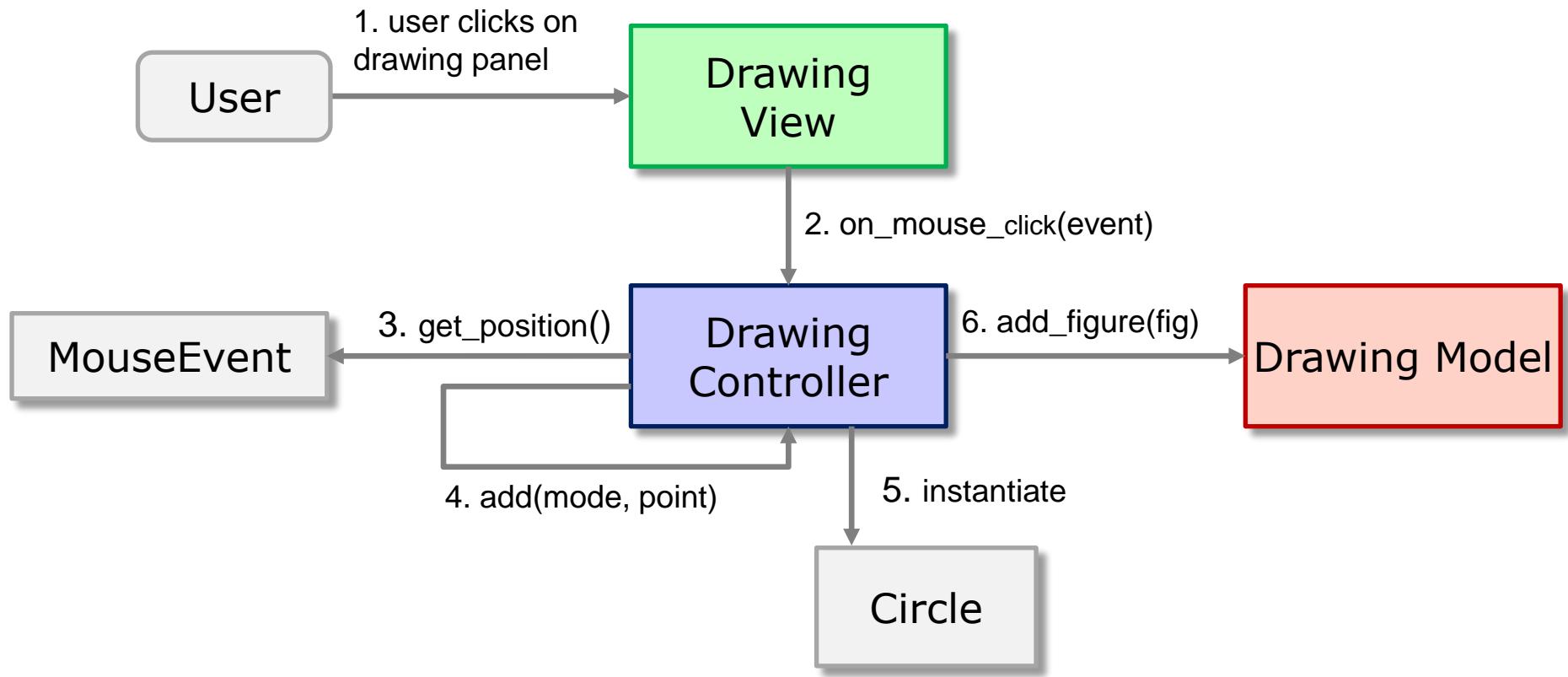
- Separates the
 - interface objects (the views)
 - from the objects which handle user input (the controllers)
 - from the application (the model).



MVC Based Applications

- MVC design pattern
 - Model
 - Business objects / Spring Beans
 - Controller
 - Handles incoming requests
 - Processing and retrieving business objects
 - One controller for every screen
 - Controller returns Model and View combination
 - View
 - Just display pages using appropriate technology

Model-View-Controller (MVC)



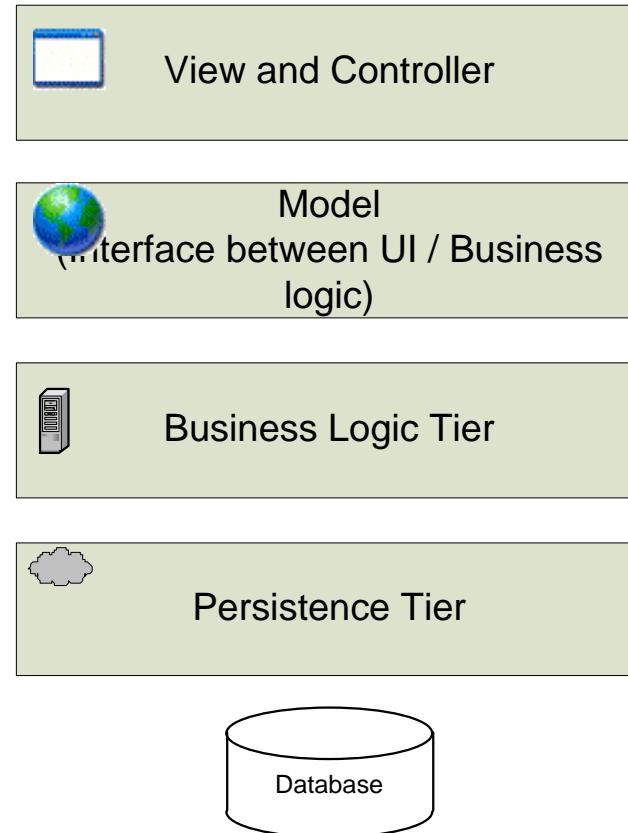
Benefits of MVC

MVC Provides for

- Reusability of application and/or user interface components
- Ability to develop the application and GUI independently
- Ability to inherit from different parts of the class hierarchy
- Ability to define different control style classes

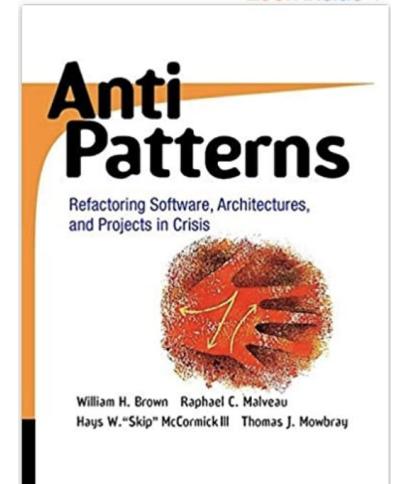
Common Web Architecture

- Persistence Tier to handle OO to Relational interface



What are Anti-Patterns?

- Commonly-reinvented bad solutions to problems
 - prevalent in non-working systems
 - identify things to avoid in the future!
- Some examples
 - Inappropriate Base Bean, Object Cesspool, Yo-Yo
 - God / Monster Object
- Textbook on anti-patterns
 - AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis,
 - by Brown, Malveau, McCormick & Mowbray
- Website on anti-patterns
 - <http://en.wikipedia.org/wiki/Anti-patterns>



God / Monster Object Anti-Pattern

- God object is an object that knows too much or does too much
- Most of a program's overall functionality is coded into a single "all-knowing" object
- Other objects within the program rely on the God object for most of their information and interaction
- Refactor by
 - Breaking up into other objects
 - Single Responsibility Principle (SRP) etc.

Generic Code Smells Checklist

- Code smell is any symptom in source code that *possibly* indicates a deeper problem

Category	Smell
Bloaters	Long Method / Function Large Class Long Parameter List Primitive Data Type Obsession Data Clumps
Object-Oriented Abusers	God object Temporary Field Refused Bequest Alternate Class with Different Interface Parallel Inheritance Hierarchies Data class

Code Smells cont'd

Category	Smell
Disposable	Lazy Class Data Class Duplicate Code Speculative Generality
Encapsulators	Message Chain Middle Man
Couplers	Feature Envy Inappropriate Intimacy
Change Preventers	Divergent Changes Shotgun Surgery
Others	Incomplete Class Library Inconsistent Comments Conditional Complexity Tests not written to same standard as standard code

Leading Figures in Patterns

- Gang of Four - GoF (Johnson et al)
 - E. Gamma, R. Helm, R. Johnson and J. Vlissades, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995, 0-201-63361-2
- Martin Fowler
 - Patterns of Enterprise Application Architecture – Addison-Wesley
- Douglas Schmidt
 - Distributed and networking patterns
- Frank Buschmann et al
 - Architectural focus
- Martin Fowler, Ken Beck, John Brant, William Opdyke & Dan Roberts
 - Refactoring: Improving the Design of Existing Code

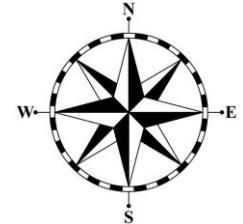
Arrays & Collection Classes



Toby Dussek

Informed Academy





Plan for Session

- Arrays
- Collections API
- ArrayList
 - Interfaces v Concrete Classes
 - Syntax
- HashMap
- Iteration and Enumeration
- Queues
- Generics and Collections
- For Loops
- Boxing and Unboxing

Creating arrays of Objects

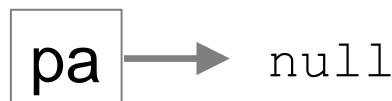
- Given the class Person
- Can create an array of Persons:

```
Person [] pa = new Person[4];
```
- Creates an array object which can hold 4 person objects
- The variable pa can hold a reference to an array object of type Person
- But there are no Person objects here!

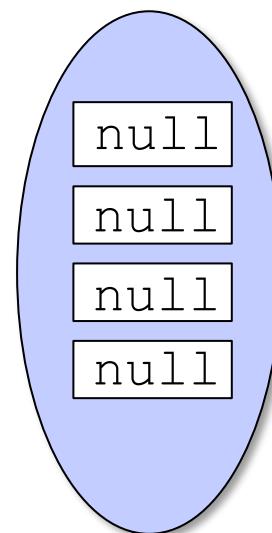
Creating Arrays of Objects

```
Person [] pa = new Person [4];
```

pa defined to
hold an array
object of type
Person[]



creates an array object
that can hold 4 Person
objects



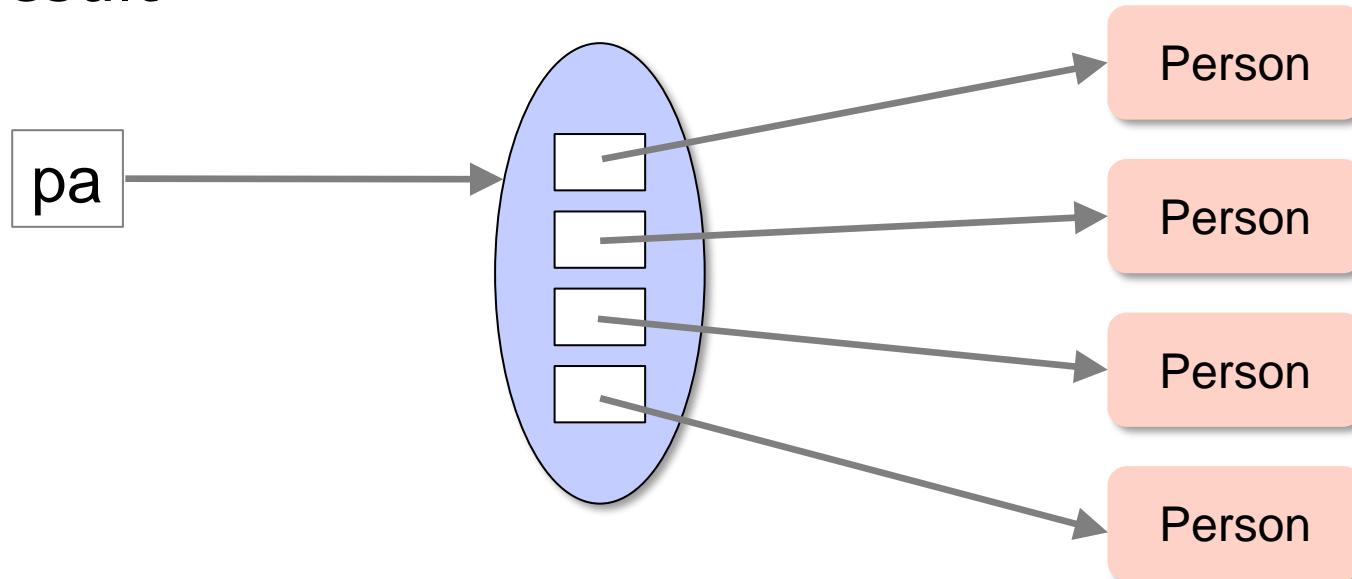
still don't have any
objects in array

Need to create
objects to hold in
array

new Person()

Creating arrays of Objects

- Final result



- Note that pa references an array object
 - array object holds references to Person objects
 - implications for passing arrays as parameters?

Accessing Array Elements

- Use square brackets with index to access array item,

```
pa[0] = new Person();  
sa[4] = "Adam";  
fa[1][1] = "Eloise";
```

- Access length of arrays

```
fa.length  
fa[1].length
```

Short hand form

- Initializing arrays can be simplified:

```
Person [] pa = {new Person(...), new Person(...)};  
String [] sa = {"John", "Denise", "Phoebe"};
```

- Can do it after declaration

```
sa = new String[]{"Phoebe", "Gryff", "Adam"};
```

- Can have multi-dimensional arrays:

```
String [][] faa = {{"John", "Denise"}, {"Paul", "Fiona"}};
```

- But what does this really mean?

- Can have ragged arrays

Collections API

- Introduced in Java 2 (SDK 1.2) for structuring data
- What is a collection?
 - A group of objects / A data structure
 - Note can only hold references to objects
- Collections data structures supported
 - Sets / SortedSets / Lists / Maps
- Implemented by different classes
 - Have different pros and cons
- Names indicate implementation approach and data structure
 - ArrayList (e.g. list implemented as an array)

Core Concrete Collection Classes

HashSet	Implements a Set
TreeSet	Intended for sorted set, will be in ascending element order
ArrayList	A simple list
LinkedList	A doubly-linked List that may provide better performance if insertion / deletion occur frequently
HashMap	A map based on a hashing approach.
TreeMap	A balanced binary tree implementation of the Map Interface. Unlike HashMap, imposes an ordering on its elements.

The ArrayList class

- Growable array of elements

```
ArrayList<String> list = new ArrayList<String>(5);
```

- <> indicate type to be held
- will be checked when element added
- referred to as Java Generics (intro'd in Java 5)

- Various utility methods

- add (object) / remove(object)
- size() / ensureCapacity() / trimToSize()
- set(index, object) / add(index, object)
- contains() / clear()
- clone() – implications of cloning

Interface v Concrete Class

- Can just use class types
- Common to have variable as interface type
 - can then hold different implementations
 - must instantiate class of course
- ArrayList example more commonly written as

```
List<String> list = new ArrayList<String>(5);
```

Diamond Operator / Syntax

□ Introduced in Java 7

- simplifies syntax when declaring generic types
- must be able to infer type

```
import java.util.ArrayList;
import java.util.List;

public class SampleApp1 {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("John");
        list.add("Denise");
        // list.add(42); // Generates a compile time error
        System.out.println(list);
    }
}
```

Hello World!

ArrayList example

```
import java.util.ArrayList;
import java.util.List;

public class SampleArrayListApp {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("John");
        list.add("Denise");
        list.add("Phoebe");
        list.add("Adam");
        System.out.println("list: " + list);
        System.out.println("list.size(): " + list.size());
        System.out.println("Contains 'John': " + list.contains("John"));
        System.out.println("Setting first element to 'Paul'");
        list.set(0, "Paul");
        System.out.println("Now Contains 'John': " + list.contains("John"));
        System.out.println("2nd Element: " + list.get(1));
        // Clone the list
        List<String> list2 = new ArrayList<>(list);
        System.out.println("list2: " + list2);
    }
}
```

list: [John, Denise, Phoebe, Adam]
list.size(): 4
Contains 'John':true
Setting first element to 'Paul'
Now Contains 'John':false
2nd Element: Denise
list2: [Paul, Denise, Phoebe, Adam]

ArrayList example

- Shorthand form for creating a list

```
package com.midmarsh;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class SampleArrayListApp2 {
    public static void main(String[] args) {
        List<String> list = Arrays.asList("John", "Denise", "Phoebe", "Adam");
        System.out.println("list: " + list);
        System.out.println("list.size(): " + list.size());
        System.out.println("Contains 'John': " + list.contains("John"));
        System.out.println("Setting first element to 'Paul'");
        list.set(0, "Paul");
        System.out.println("Now Contains 'John': " + list.contains("John"));
        System.out.println("2nd Element: " + list.get(1));
        // Clone the list
        List<String> list2 = new ArrayList<>(list);
        System.out.println("list2: " + list2);
    }
}
```

list: [John, Denise, Phoebe, Adam]
list.size(): 4
Contains 'John':true
Setting first element to 'Paul'
Now Contains 'John':false
2nd Element: Denise
list2: [Paul, Denise, Phoebe, Adam]

The HashMap class

- Maps keys to elements

```
HashMap<String, String> map = new HashMap<String, String>();
```

- Variety of methods

- void put(key, element)
- Object get(key)
- remove(key)
- int size()
- containsValue(object) / containsKey(key)
- keySet() / values()

Sample HashMap

example

```
import java.util.Collection;
import java.util.HashMap;
import java.util.Map;
import java.util.Set;
```

```
public class SampleMapApp {
    public static void main(String[] args) {
        Map<String, String> map = new HashMap<>();
        map.put("John", "A5");
        map.put("Denise", "B57");
        map.put("Jason", "C5");
        map.put("Carol", "B56");
        System.out.println("Map: " + map);
        System.out.println("map.size(): " + map.size());
        System.out.println("map.get('John'): " + map.get("John"));
        System.out.println("Contains key 'Denise': " + map.containsKey("Denise"));
        System.out.println("Map contains value C5: " + map.containsValue("C5"));
        Set<String> keys = map.keySet();
        System.out.println("Keys: " + keys);
        Collection<String> values = map.values();
        System.out.println("Values: " + values);
    }
}
```

```
Map: {Denise=B57, John=A5, Jason=C5, Carol=B56}
map.size(): 4
map.get('John'): A5
Contains key 'Denise': true
Map contains value C5: true
Keys: [Denise, John, Jason, Carol]
Values: [B57, A5, C5, B56]
```

Iterations

- Set of objects for iteration
- Lists, Sets have
 - iterator()
- Maps have
 - get keySet / values
 - iterate over these

```
package com.midmarsh;  
  
import java.util.Arrays;  
import java.util.Iterator;  
import java.util.List;  
  
public class SampleIterationApp {  
    public static void main(String[] args) {  
        List<String> list =  
            Arrays.asList("John", "Denise",  
                         "Phoebe", "Adam");  
        // Iterate over list  
        Iterator<String> it = list.iterator();  
        while (it.hasNext()) {  
            String s = it.next();  
            System.out.println(s);  
        }  
    }  
}
```

John
Denise
Phoebe
Adam

Iterations 2

- Can also use an Enumeration
 - pre dates iterators
- Very like an iterator

```
Enumeration e = list.elements();
while (e.hasMoreElements()) {
    System.out.println(e.nextElement());
}
```

- Why have two?
 - Enumeration was the original approach
 - some libraries still use it
- But Iterators more flexible

Generics and Collections

- A couple of points to note:

```
List<String> list = new ArrayList<String>();
```

- List and ArrayList are marked as only being able to hold strings
- That the result of it.next() is a String
- any iterator obtained from list knows it will return Strings

- Also note these are **Typed Collections**

- thus both of the following:

```
List<Integer> intList = new ArrayList<Integer>();
```

```
List<String> stringList = new ArrayList<String>();
```

- Produce instances of ArrayList –
 - but are parameterised appropriately

For loop and Collections

- Iteration over collections introduced in Java 5

```
import java.util.Arrays;
import java.util.List;

public class SampleForLoopApp {
    public static void main(String[] args) {
        List<String> list = Arrays.asList("John", "Denise",
                                         "Phoebe", "Adam");
        System.out.println("list: " + list);
        for (String s : list) {
            System.out.println(s);
        }
    }
}
```

```
list: [John, Denise, Phoebe, Adam]
John
Denise
Phoebe
Adam
```

- When you see the colon (:) read it as “in”
 - Above reads as “for each String s in list”

Boxing and UnBoxing

- Introduced in Java 5
- Ability for JVM to convert between basic and equivalent object types
 - e.g. int to Integer (and vice versa)
 - e.g. char to Character
- Actually wraps basic type in an object
 - Referred to as boxing – when wrapping
 - Unboxing – when unwrapping

Autoboxing with Collections

- ❑ Use reference types when creating collections
 - can add and retrieve primitive types
 - internally boxes to object wrappers

```
public class SampleBoxingApp {  
    public static void main(String[] args) {  
        int count = 0;  
        List<Integer> list = new ArrayList<>();  
        list.add(2);  
        list.add(3);  
        System.out.println("list: " + list);  
        for (int i : list) {  
            count = count + i;  
        }  
        System.out.println("Count: " + count);  
    }  
}
```

What is actually held in the list?

list: [2, 3]
Count: 5

- Autoboxing hides conversions

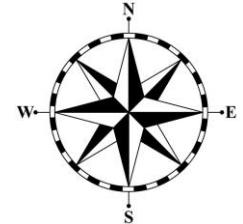
Functions



Toby Dussek

Informed Academy





Plan for Session

- Functional Interfaces
- Using Functional Interfaces
- Lambdas in Java
- Closing / Capturing Variables
- Combining Functions
- Higher Order Functions
- Method References



Functional Interfaces

- Introduced Functional Interfaces
 - explicit representation of functional behavioural style
 - understood by compiler
- Core features in
 - `java.util.function package`

```
package java.util.function

@FunctionalInterface
public interface Function<A,R> {
    R apply ( A arg );
}
```

Functional Interfaces

- Must have a *single* abstract method
 - used to invoke the *functional* behaviour
- Can have any number of static and default methods

```
package java.util.function

@FunctionalInterface
public interface Function<A,R> {

    R apply ( A arg )

    // Any default or static methods
}
```

Functional Interfaces

- Numerous functional interfaces available
 - `Predicate<T>`
 - Represents a predicate with 1 arg
 - `Supplier<T>`
 - Represents a supplier of results
 - `Consumer<T>`
 - Represents an operation that accepts a single input argument and returns no result
 - Plus convenience versions of above
 - e.g. `IntSupplier`

Using Functional Interfaces

- Can define functional class to double an integer

```
class Doubler implements Function<Integer, Integer> {  
  
    @Override  
    public Integer apply(Integer t) {  
        return t * 2;  
    }  
  
}
```

- Note
 - implements the Function<A, R> interface
 - A will be an Integer
 - R will be an Integer
 - Implements the apply method with appropriate types

Lambdas in Java

- Creating instances of Function objects verbose and somewhat counter intuitive
- Java has shorthand form
 - *lambda* expressions
 - used to create a *function literal*
- Lambda function instances
 - are still instances of Functional Interfaces
 - but a lot simpler to write

Lambdas in Java

- Defined by

- (arguments) -> function body
 - for example

```
( Integer i ) -> i * 2
```

Argument

Function<Integer, Integer>

Result of expression returned

- in fact, if only one argument don't even need brackets
 - if type can be inferred

```
Function<Integer, Integer> squareMe = i -> i * i;
```

- notice the type of the variable squareMe

Lambdas in Java

- Lambdas represent function objects
 - but under the hood more efficient
 - don't always need to create a traditional object
 - can effectively in-line lambdas
- Be careful of using the "this" reference
 - because of above `this` can reference enclosing object
 - not the lambda

```
public class SampleFuncApp3 {  
    Runnable func = () -> System.out.println(this);  
  
    public static void main(String[] args) {  
        SampleFuncApp3 app = new SampleFuncApp3();  
        app.func.run();  
    }  
}
```

SampleFuncApp3@4d7e1886
282

Closing / Capturing variables

- Functions can capture variables from within their context
- Local variables must be final
 - or *effectively* final – i.e. not changed after definition

```
public static void main(String[] args) {  
    // Must be final (or effectively final) - can't change it  
    int MAX = 100;  
    // The Lambda captures the value of MAX  
    Predicate<Integer> check = i -> i <= MAX;  
    System.out.println(check.test(10));  
}
```

- Instance variables of enclosing object
 - do not need to be final as enclosing object can be captured

```
class Checker {  
    // Does not need to be final (or effectively final)  
    public int MAX = 100;  
    // Lambda captures MAX but w.r.t. enclosing object  
    public Predicate<Integer> check = i -> i <= MAX;  
}
```

Combining Functions

- Composition of functions
 - allows new functions to be created by combining together existing functions
 - the output of one becomes the input to the next
- Utilities provided as default methods
 - in `Function<A, R>` interface
 - given functions 'f' and 'g'
 - `f.compose(g)` is g followed by f
 - `f.andThen(g)` is f followed by g
 - use `apply()` type methods with result

Combining Functions

- Examples of function composition

```
func1(2) = 4
func2(2) = 6
(func1 compose func2)(2) = 8
(func1 andThen func2)(2) = 12
func3(2) = 12
```

```
import java.util.function.*;

public class SampleFuncApp6 {
    public static void main(String[] args) {
        Function<Integer, Integer> func1 = i -> i + 2;
        Function<Integer, Integer> func2 = i -> i * 3;
        // Basic use of functions
        System.out.println("func1(2) = " + func1.apply(2));
        System.out.println("func2(2) = " + func2.apply(2));
        // Composition versus andThen
        // Composition is f2 followed by f1
        System.out.println("(func1 compose func2)(2) = " +
                           (func1.compose(func2)).apply(2));
        // And Then is f1 and then f2
        System.out.println("(func1 andThen func2)(2) = " +
                           (func1.andThen(func2)).apply(2));
        // Actually create a new function using these constructs
        Function<Integer, Integer> func3 = func1.andThen(func2);
        System.out.println("func3(2) = " + func3.apply(2));
    }
}
```

Higher Order Functions

- Are functions that
 - either take one or more functions as a parameter
 - return a function as a result or do both
- Can pass apply a function that will be *applied*

```
// Example of a higher order function - a function that takes a function
class Processor implements Function<Function<Integer, Integer>, Integer> {
    private int value = 0;

    public Integer apply(Function<Integer, Integer> func) {
        return func.apply(value);
    }

    public void setValue(int value) {
        this.value = value;
    }
}
```

```
Processor p = new Processor();
p.setValue(5);
```

```
System.out.println(p.apply(i -> i * i));
System.out.println(p.apply(i -> i * 2));
```

25
10

286

Method References

- Allows constructors and methods to be referenced
 - without executing them
 - can then execute them at a future point in time if required
- Four Types of Method Reference
 - reference to a *static* method
 - reference to a *constructor*
 - reference to a *method on a specific instance*
 - reference to a *method* on an object of a *particular type*
- Method References are not parameterized by values
 - instead, use a factory method or a lambda

Method References

- Given a class such as Person
 - with static method
 - constructor
 - instance methods

```
class Person {  
    private static int count = 0;  
  
    private String name = "";  
  
    public static int increment() {  
        count = count + 1;  
        return count;  
    }  
  
    public Person(String name) {  
        this.name = name;  
    }  
  
    public String getName() {  
        return name;  
    }  
}
```

Method References

- Reference to a static method
 - Person::increment
- Reference to instance method of a specific object
 - p::getName
- Reference to instance method of any object of class
 - Person::getName
- Reference to constructor
 - ClassName::new

Method References

```
import java.util.function.*;

public class SampleMRApp {
    public static void main(String[] args) {
        Person p = new Person("Bob");
        // Static Ref
        Supplier<Integer> staticRef = Person::increment;
        System.out.println("staticRef.get(): " + staticRef.get());
        // Constructor Ref
        Function<String, Person> consRef = Person::new;
        Person p2 = consRef.apply("Jasmine");
        System.out.println("Person p2: " + p2);
        // Specific Instance
        Supplier<String> objRef = p::getName;
        System.out.println("objRef.get(): " + objRef.get());
        // Any instance
        Function<Person, String> anyRef = Person::getName;
        System.out.println("anyRef.apply(p2): " + anyRef.apply(p2));
    }
}
```

```
staticRef.get(): 1
Person p2: Person [name=Jasmine]
objRef.get(): Bob
anyRef.apply(p2): Jasmine
```

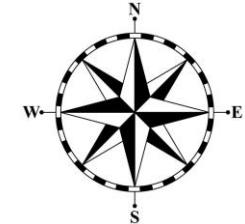
Java Streams



Toby Dussek

Informed Academy





Plan for Session

- Streams
- Stream Chains
- Stream Concepts
- Streams from Collections
- Terminal / Non Terminal Ops
- Creating a Stream
- Map Operation
- Collectors
- Filter, Sorted, ForEach Operations
- Not Just Collections



Streams

- Java8 introduced a Streams API
- Supports functional-style operations on streams of elements

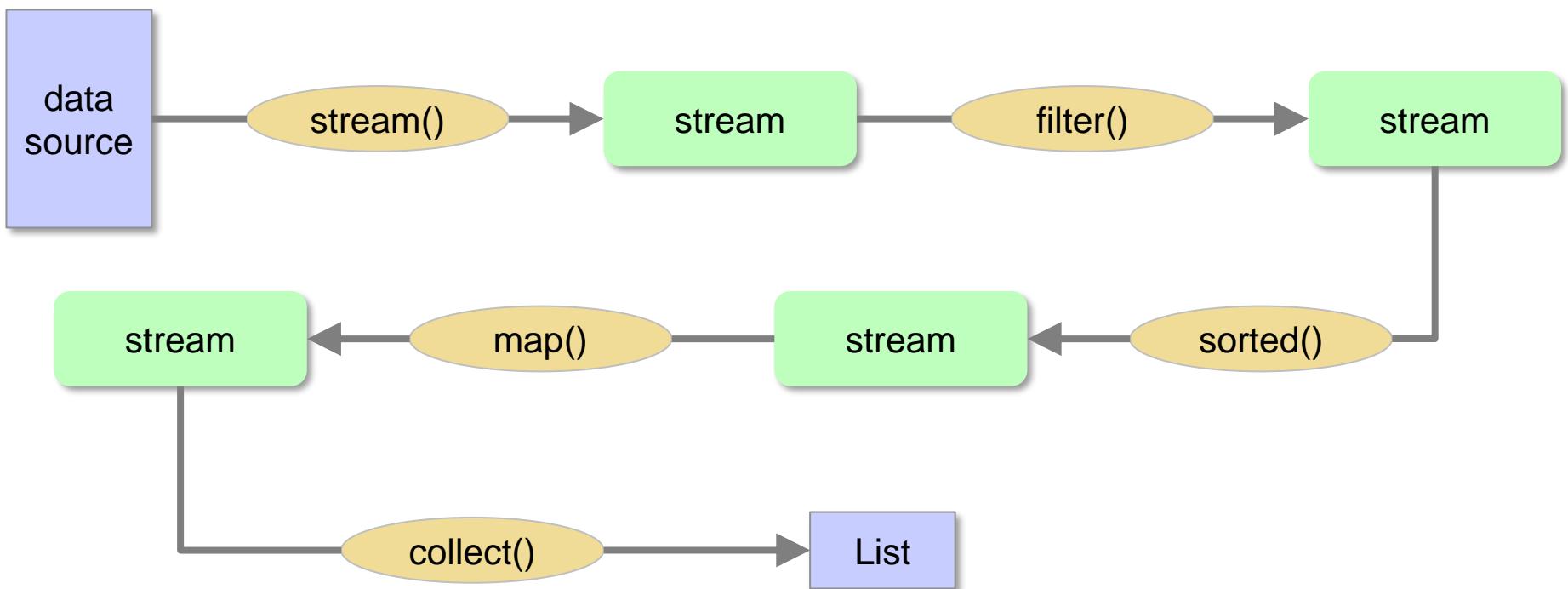
```
List<Trade> trades = new ArrayList<>();  
...  
System.out.println(  
    trades.stream()  
        .map(t -> t.getQuantity())  
        .max(Integer::max));
```

- Integrated into the Collections API
- Can apply bulk operations to contents of collections etc.



Stream Chains

- Stream can work sequentially or in parallel
- Can connect output of one stream to input of another stream
- Can have a chain and collect results



Stream Concepts

- Can work with infinite data streams
- Have a functional style to them
 - no side effects, don't modify the source
 - often use higher order functions
- Operations can be pipelined together
 - and are typically executed lazily
- Streams are an extension to collections
 - key methods added to collections
 - `stream()` creates a (sequential) stream object
 - `parallelStream()` creates a parallel processing stream
 - collection must be treated as immutable

Streams from Collections

- Can use functional operations on the stream
 - `map()` applies a function to each element in stream and returns a stream
 - `filter()` filters contents of stream and returns a stream
 - `flatMap()` apply a map function and flatten result
 - `reduce()` elements in stream to a single value
 - `forEach()` applies operation to element element in stream.
Does not return a stream
 - `collect()` ends the stream process and transforms stream data (e.g. into a List)

Streams from Collections

□ Other supported methods

- `count()` number of elements in stream
- `limit()` only pass first n elements to the next stream
- `sorted()` – sorts the items in the stream
- `max()` / `min()` returns the maximum or minimum element in the stream
- `distinct()` ensures all values in the stream are distinct

Terminal / Non Terminal Ops

- Two categories of operation
 - terminal or intermediate
- Terminal operations
 - return `void` or a non stream object such as a `List`
 - examples include `collect` and `forEach`
- Intermediate (or non terminal operations)
 - return another stream
 - type inferred from result of operation
 - allows for chaining of operations

Students Example

- Student simple value object style class

```
public class Student {  
    private final String subject;  
    private int grade;  
    private final String name;  
    public Student(String subject, int grade, String name) {  
        this.subject = subject;  
        this.grade = grade;  
        this.name = name;  
    }  
    public String getSubject() {  
        return subject;  
    }  
    public int getGrade() { return grade; }  
    public void setGrade(int grade) {  
        this.grade = grade;  
    }  
    public String getName() { return name; }  
    public String toString() {  
        return "Student[id=" + grade + ")" + name + ", subject=" + subject + "]";  
    }  
}
```

```
List<Student> students = Arrays.asList(  
    new Student("History", 65, "Gryff"),  
    new Student("English", 75, "Jasmine"),  
    new Student("Pharmacology", 68, "Adam"),  
    new Student("Law", 63, "Eloise"));
```

Creating a Stream

- Given the collection `students`
 - can obtain a stream using
`students.stream()`
- Can then apply operators to the stream
 - for example, apply a function to each element in the stream
`students.stream().map(t -> t.getSubject())`
- Then collect results

```
List<String> subjects = students.stream().map(t -> t.getSubject())
.collect(Collectors.toList());
```
- Or count results

```
students.stream().map(t -> t.getGrade()).count();
```
- Or find max values returned

```
students.stream().map(t -> t.getGrade()).max(Comparator.naturalOrder())
```

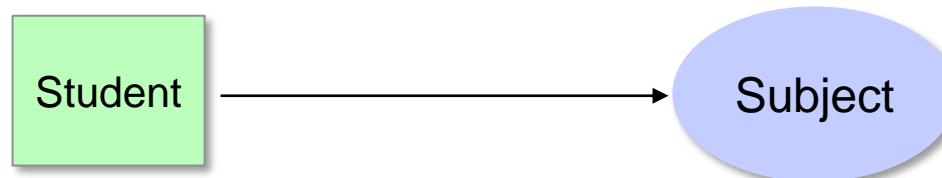
Creating a Stream

```
[History, English, Pharmacology, Law]  
4  
65
```

```
import java.util.Arrays;  
import java.util.List;  
import java.util.stream.Collectors;  
  
public class StreamsExample1 {  
    public static void main(String[] args) {  
        List<Student> students =  
            Arrays.asList(new Student("History", 65, "Gryff"),  
                         new Student("English", 75, "Jasmine"),  
                         new Student("Pharmacology", 68, "Adam"),  
                         new Student("Law", 63, "Eloise"));  
  
        List<String> subjects = students.stream()  
            .map(t -> t.getSubject())  
            .collect(Collectors.toList());  
        System.out.println(subjects);  
  
        long temp = students.stream().map(t -> t.getGrade()).count();  
        System.out.println(temp);  
  
        students.stream()  
            .map(t -> t.getGrade())  
            .max(Comparator.naturalOrder())  
            .ifPresent(System.out::println);  
    }  
}
```

Map Operation

- Applies / Maps a function
 - to each element in stream
 - non terminal operation
 - function returns a result
 - result is passed on
 - function can transform an element
 - example transforms a Student object to the corresponding subject



- Compare with a for loop

Collectors

- collect operation
 - terminal operation used to transform stream elements into lists, sets, maps etc.
- Various built-in collector classes available
 - `Collectors.toSet()`
 - `Collectors.toList()`
 - `Collectors.groupingBy(func)` provides `Map<key, List>`
 - `Collectors.averagingInt(func)`
- Can implement your own
 - `java.util.stream.Collector`

Filter Operation

- Can be used to select elements of interest from stream
 - predicate function must return True to allow element to pass on
 - for example select all students with grades over 69

```
List<Student> symbols =  
    students.stream().filter(t -> t.getGrade() > 69)  
        .collect(Collectors.toList());
```

```
System.out.println(symbols);
```

```
[Student[(id=75)Jasmine, subject=English]]
```

Sorted Operation

- Can sort output from a stream
 - uses the Comparator.comparing higher order function function

```
students.stream()  
  .sorted(comparing(Student::getGrade))  
  .forEach(System.out::println);
```

```
Student[(id=63)Eloise, subject=Law]  
Student[(id=65)Gryff, subject=History]  
Student[(id=68)Adam, subject=Pharmacology]  
Student[(id=75)Jasmine, subject=English]
```

- can also sort a collection in this way

```
students.sort(comparing(Trade::getGrade));
```

ForEach

- ❑ Located on Iterable interface and on Stream
 - ❑ terminal operator
 - ❑ operates on each element in stream

```
students.stream().forEach(System.out::println);
```

```
students.stream().forEach(s -> s.setGrade(s.getGrade() + 10));
```

```
students.stream().forEach(System.out::println);
```

```
Student[(id=65)Gryff, subject=History]
Student[(id=75)Jasmine, subject=English]
Student[(id=68)Adam, subject=Pharmacology]
Student[(id=63)Eloise, subject=Law]
Student[(id=75)Gryff, subject=History]
Student[(id=85)Jasmine, subject=English]
Student[(id=78)Adam, subject=Pharmacology]
Student[(id=73)Eloise, subject=Law]
```

- ❑ Could also call directly on students

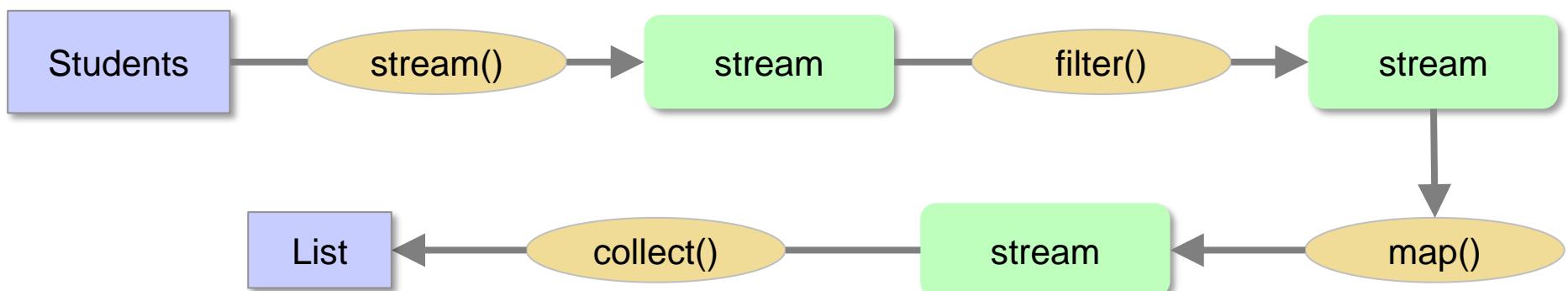
```
students.forEach(System.out::println);
```

Pipelining Operations

- Can have multiple operations pipelined together

```
List<String> subjects =  
students.stream()  
.filter(t -> t.getGrade() > 30)  
.map(Student::getSubject)  
.collect(Collectors.toList());  
  
System.out.println(subjects);
```

[History, English, Pharmacology, Law]



Not Just Collections

- It is not necessary to have a collection to use streams
- Can use Stream.of method to create a stream
 - takes a group of object references

```
Stream.of("a1", "a2", "a3")
    .findFirst()
    .ifPresent(System.out::println);
```

a1

- Special types of streams for primitive types

- e.g. IntStream, LongStream and DoubleStream

```
IntStream.range(1, 4)
    .forEach(System.out::println);
```

1
2
3

```
IntStream.generate(
    () -> (int)(Math.random() * 100))
    .limit(10).forEach(System.out::println);
```

Java Error and Exception Handling

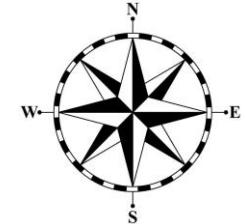


Toby Dussek

Informed Academy



Plan for Session



- Errors & Exceptions
- Part of the Exception Hierarchy
- Exception Handling
- Local Handling
- Exception Handling Example
- Passing the Buck
- Try with Resources
- Defining new Exceptions
- Chained Exceptions
- Exception Safe Code

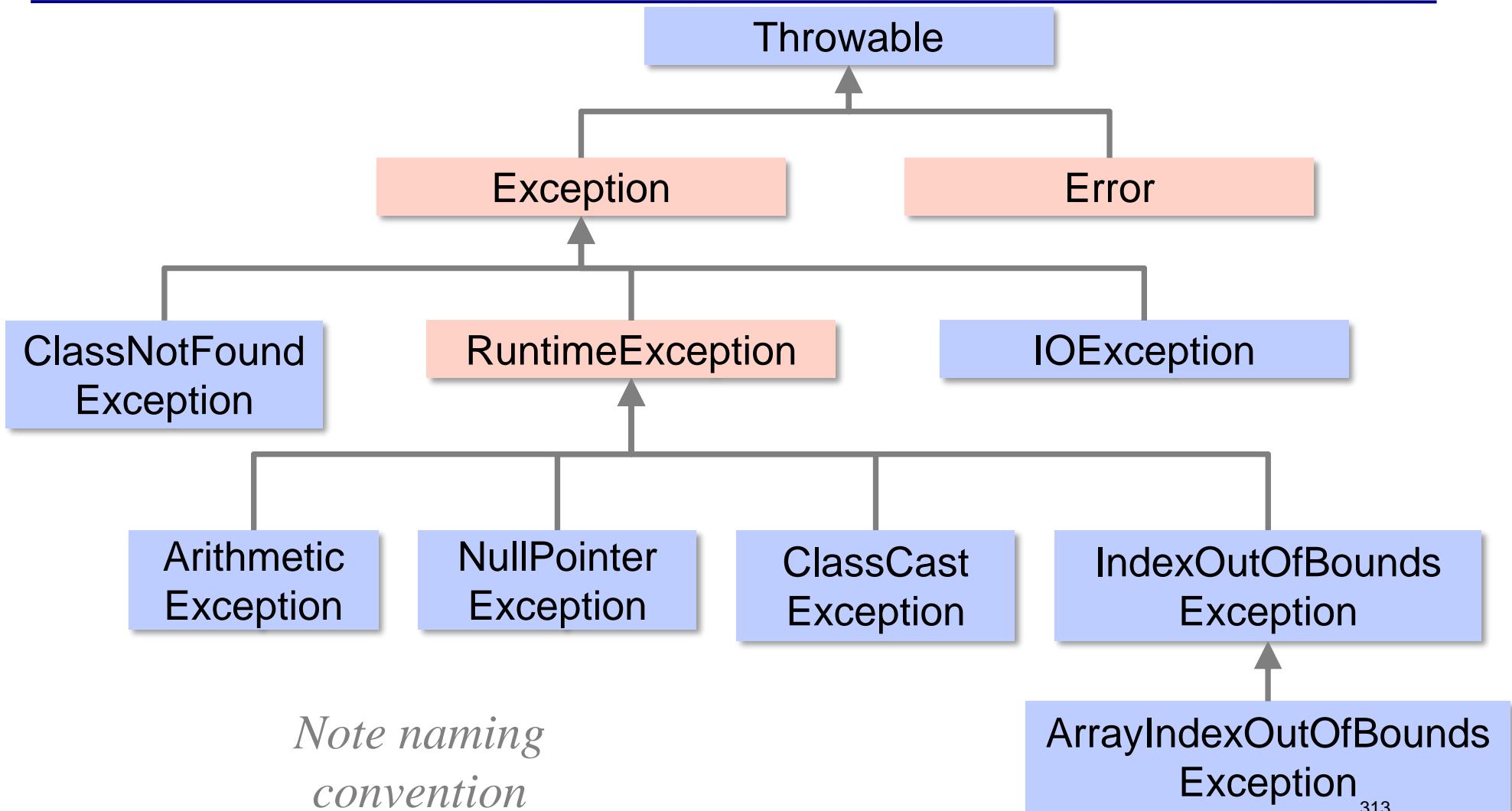
Errors & Exceptions

- Exceptions and Errors are objects in Java
- Can occur in a variety of situations
 - File does not exist
 - Network connection fails
 - Class definition is missing at runtime
 - Out of memory
 - An array index is out of range
- Want to be notified about these and take appropriate action

Exceptions Types in Java

- See the `java.lang.Throwable` class
- Also see `java.lang.Exception`
- Three types of Exception in Java
 - Error - catastrophic error
 - Exception - should be handled
 - (may occur in a correctly functioning pgm)
 - E.g. `IOException`
 - Runtime - bugs in code
 - E.g. `NullPointerException`
 - E.g. `ArrayIndexOutOfBoundsException`

Part of Exception Hierarchy



Exception Handling

- Based on C++
- Exception raised by instantiating an exception class
- If a Runtime exception can handle
- If an Exception then must handle
- Must say how you will handle exceptions
 - try { } catch { } blocks
 - throws clause

Local Handling

- Uses the `try { } catch { }` block
- Can have 1 or more catch blocks
 - each tried in turn
- Try block must be capable of raising specified exception(s)
- Catches exception or subclass
- Try blocks may be nested
- `finally { }` block (optional) always runs

Exception Handling Example

```
15 ⊖  public void save(String filename) {  
16      System.out.println("Entering save() method");  
17      try {  
18          System.out.println("Starting to save data to file");  
19          saveDataToFile(filename);  
20          System.out.println("Completed saving data");  
21      } catch (IOException e) {  
22          e.printStackTrace();  
23      } catch (Exception e) {  
24          System.out.println("Something went wrong");  
25      }  
26      System.out.println("Exiting save() method");  
27  }
```

- Enter try-catch block line 17
- If all goes okay lines 18-20 execute
- If an exception at line 19
 - Line 21 check exception
 - if IOException run line 22
 - Otherwise look at 23 - if type of Exception run line 24
- Run line 26 if all ok or an exception raised

Passing the buck

- Can pass exception handling responsibility
- Use the throws clause on method specification for Managed Exceptions

```
29@    private void saveDataToFile(String filename) throws IOException {  
30        FileWriter fw = new FileWriter(filename);  
31        BufferedWriter bw = new BufferedWriter(fw);  
32        PrintWriter pw = new PrintWriter(bw);  
33        pw.print("hello world");  
34        pw.close();  
35    }
```

- Anything that calls saveDataToFile must handle the IOException
 - or throw exception back up call hierarchy

Defining new Exceptions

- Exceptions are classes
 - so can create a new Exception type by defining a new class
- Must subclass an existing exception type
 - for it to work with exception handling infrastructure
 - just need to find correct parent class
- Raise an exception using `new`
- Throw an exception using `throw`
- Note exception details generated where created

Example Exception

```
public class SetException extends RuntimeException {  
    public int code;  
  
    public SetException(String cause, int code) {  
        super(cause);  
        this.code = code;  
    }  
  
    public SetException(Exception exp, String cause, int code) {  
        super(cause, exp);  
        this.code = code;  
    }  
}
```

- To use create a new instance and throw SetException

```
throw new SetException("Already in Set", 143);
```

Chained Exceptions

❑ Exception handling

- When exception occurs you:
 - 1) deal with the exception
 - 2) throw it on up the stack (if the method signature permits)
 - 3) create a new exception and throw that instead
- But for option 3, often the original context may be lost

❑ Chained exceptions

- New exception chained to original
- Prevents loss of original exception information
 - stack trace, type, error message

❑ Since Java 1.4 all Throwables can be chained

- new constructors:

- `public Throwable(String msg, Throwable cause)`

Chained Exception example

- Catch original exception & convert into application specific exp

```
import java.io.IOException;
import org.w3c.dom.Document;
import org.xml.sax.SAXException;

public class XMLUtilities {
    protected Document getEmptyDocument() {
        Document doc = null;
        try {
            doc = DOMFactory.getDocument();
        } catch (IOException exp) {
            throw new DataAccessManagerException(exp.getMessage(), exp);
        } catch (SAXException exp) {
            throw new DataAccessManagerException(exp.getMessage(), exp);
        }
        return doc;
    }
}
```

Testing

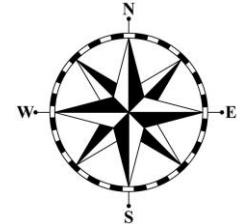


Toby Dussek

Informed Academy



framework training
business value through education



Plan for Session

- Motivation
- Reality Check
- Classification of Tests
- Black and White Box Testing
- V-Shaped Model of Testing
- Levels of test
- Who Tests Software
- Test Driven Development
- Different Test Environments

Motivation

- Verification in computing typically accomplished by running test cases
- Tests are there to ensure
 - correct functionality of modules, subsystems, or the system as a whole
 - that the system is usable
 - that performance is acceptable
 - that it is stable and reliable over a given period of time and under particularly intense load

Reality Check

- Dijkstra said, “Program testing can be used to show the presence of bugs, but never to show their absence.”
 - i.e. it is not possible to check every line of code/every optional path in a large system
- Results of testing is not a yes/no response
 - You must assess results
 - Errors in large systems are unavoidable
 - Some errors are left and are “tolerable”
 - Others are critical and must be repaired

Classification of Tests

- ❑ There are two levels of classification
 - One distinguishes at level of granularity
 - Unit Test
 - Integration test
 - Sub-system test
 - System test
 - One distinguishes based on methodology
 - Black Box (Functional) testing
 - Treats as a black box – what it should do
 - White Box (Structural) testing
 - Treat as a clear box – how it does it – look at the mechanism

Black & White Box Testing

- White Box (aka Structural) Testing
 - Testing software using information about its internal structure and implementation
 - Tests what the software actually does
 - Performed incrementally during creation of code
 - Relies on statements and structure of code itself and is run by the developer team

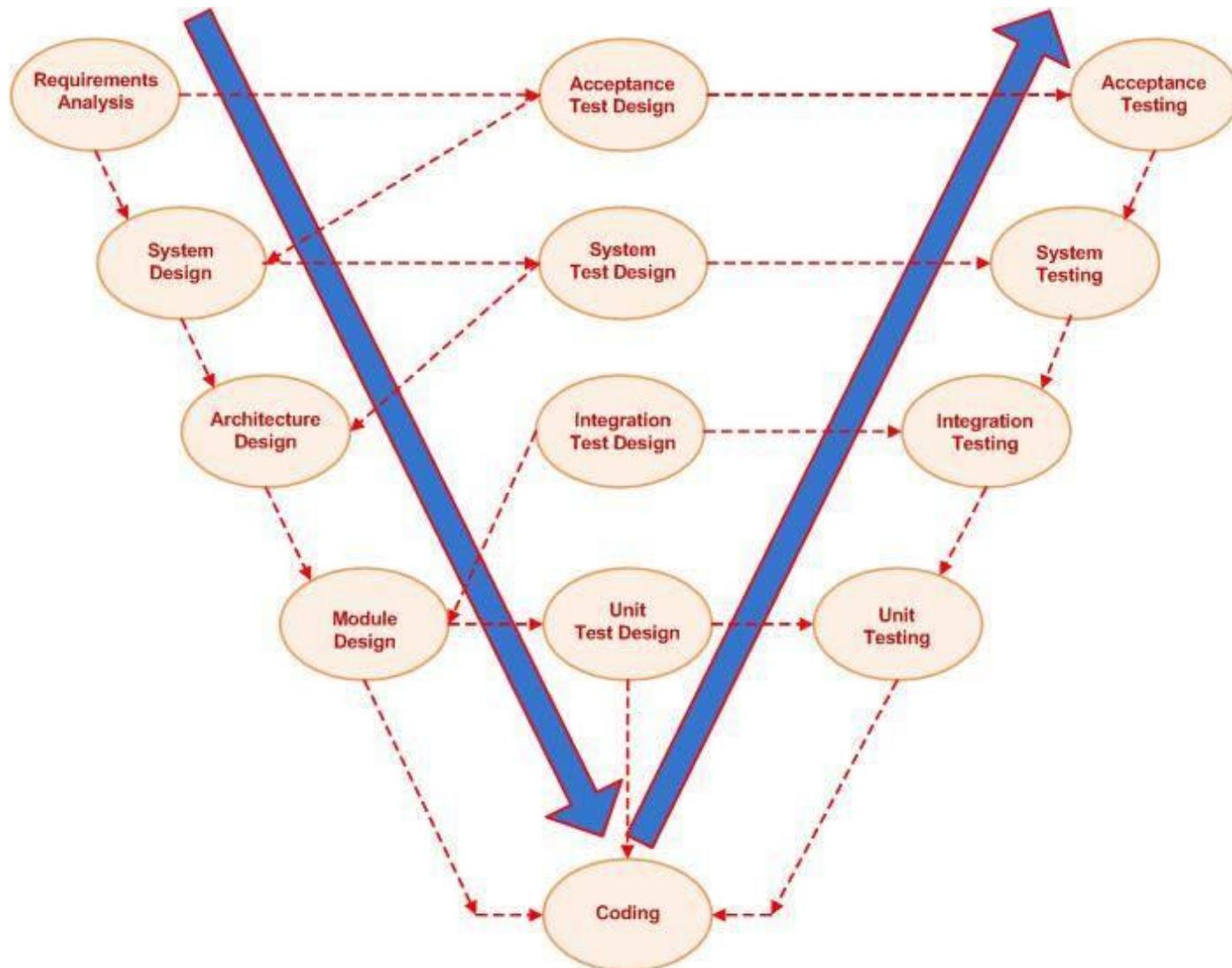
- Black Box (aka Functional) Testing
 - Testing without reference to implementation
 - Evaluated against the specification
 - Tests what the product is supposed to do
 - Run by software engineers and domain users

The V-Shaped Model of Testing (1)

- A (traditional) structured approach to testing
- Testing is emphasized in this model
- Indicates relationship between testing & development
- Highlights the different scopes of testing
 - From low level unit tests (Testing in the Small)
 - To integration and system testing
 - In its various forms (functional, performance, stability, acceptance, etc.)

Its strictly linear approach does not lend itself well to TDD but it is useful for the breadth of testing types it describes

The V-Shaped Model of Testing (2)



Levels of test

- Unit testing
 - Testing individual elements – may be at the method level or individual class level
- Integration Testing
 - Testing combinations of parts to determine they function together correctly
- Sub-system testing
 - Essentially higher level of integration tests – larger elements of the system together
- System testing
 - Black box type testing that is based on overall requirements specifications; covers all combined parts of a system

Levels of test

- Smoke tests
 - AKA Sanity testing
 - A sub set of systems tests used to provide indicative information on the likely suitability of a system for further testing / stability
- Acceptance testing
 - Testing of the whole system relative to previously agreed acceptance criteria
 - Typically relate to functional requirements
 - May also include Non Functional Requirements e.g. performance
 - May incorporate UAT or have UAT as a separate process
- Functional testing
 - Black box type testing geared to functional requirements

Levels of test

- Installation Testing
 - Testing the installation process and the end result
- Upgrade Testing
 - Testing the upgrade process to ensure that it can be done reliably and in a timely manner
- Regression Testing
 - Testing that any changes made do not affect previous functionality
- Stability Testing
 - Verifying that system works at average load with no unexpected issues for a required period of time
- Reliability Testing
 - Measuring the occurrence of failure
- Compatibility testing
 - For different hardware/software/operating system etc.

Levels of test

- Performance testing
 - Average / Worst case response time
 - Memory profile
 - Load testing at expected through puts
 - CPU utilization etc.
- Stress testing
 - Stress-testing the system with maximum load
- Usability testing
 - Often assessment by proxy e.g. number of times users entered data incorrectly – often more subjective
- Security testing
 - May involve penetration testing/ security review etc.

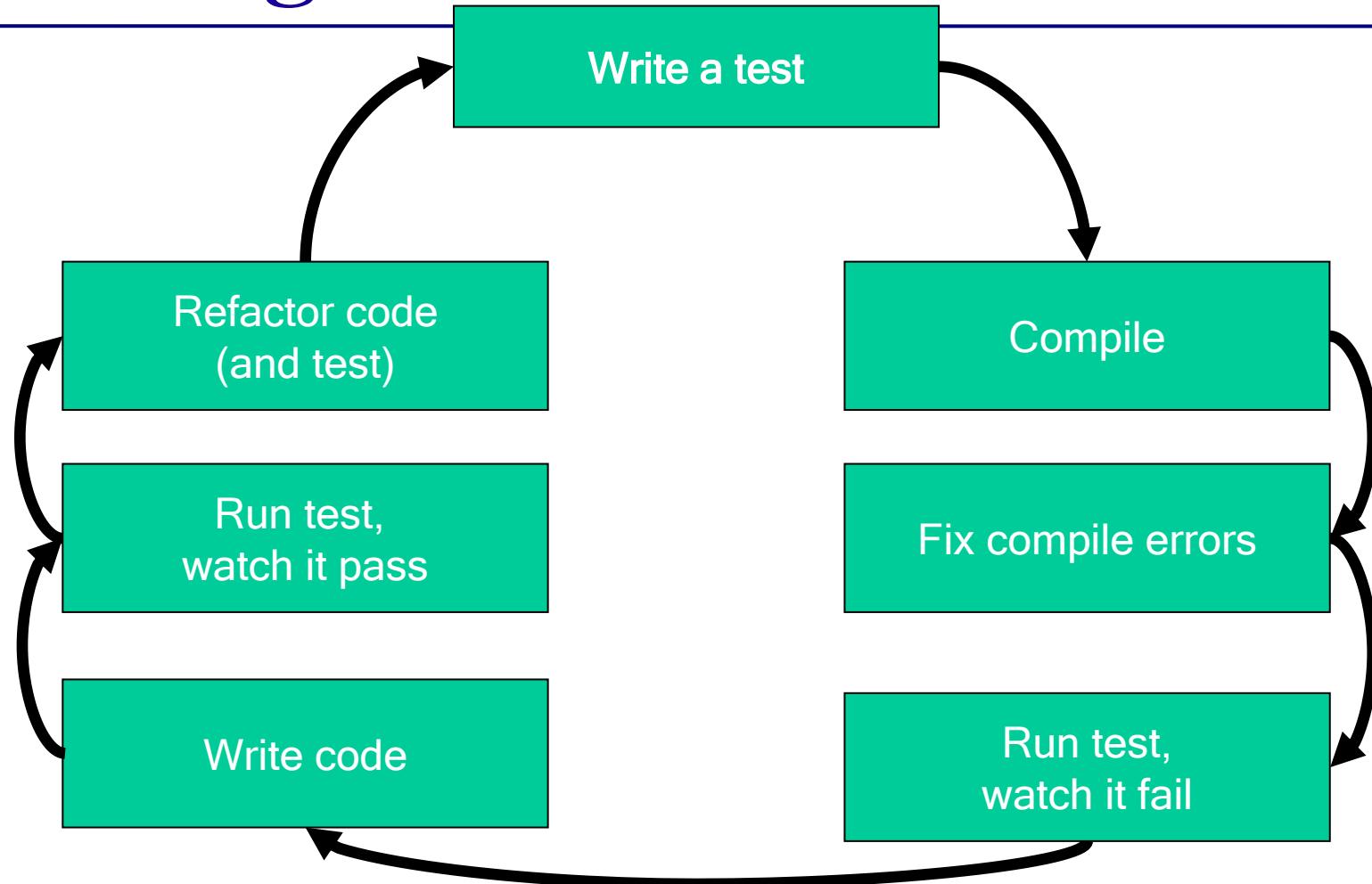
Who Tests Software?

- ❑ Developers
 - Typically at Unit, Integration, Sub-system Level
- ❑ Testers (may include Developers)
 - Often at System, Acceptance, Performance Level
- ❑ Operations
 - Including Installation / Deployment, Upgrade
- ❑ Users
 - UAT, Acceptance etc.
- ❑ Security specialists
- ❑ Regulatory authorities

What is TDD?

- TDD is a technique whereby you write your test cases **before** you write any implementation code
- Tests drive or dictate the code that is developed
- An indication of “intent”
 - Tests provide a specification of “what” a piece of code actually does
 - Some might argue that “tests are part of the documentation”

TDD Stages



Why TDD?

- Programmers dislike testing
 - They will test reasonably thoroughly the first time
 - The second time however, testing is usually less thorough
 - The third time, well..
- Testing is considered a “boring” task
- Testing might be the job of another department / person
- TDD encourages programmers to maintain an exhaustive set of repeatable tests
 - Tests live alongside the Class/Code Under Test (CUT)
 - With tool support, tests can be run selectively
 - The tests can be run after every single change

TDD: Design Evolution

- TDD is more about design evolution
- The act of writing a unit test is more an act of design than of verification
- In writing the test you are specifying what the software should do
- You then implement that required behaviour and any changes this requires to existing code
- TDD is thus Test First Design + Refactoring

Testing the Build

- Make your build Self Testing
 - A suite of automated tests
- Provide appropriate tests at appropriate levels
 - Unit tests, integration tests, smoke tests, system tests
- Ensure tests are self defining
 - Provide all they need to run the tests
- To maximise the Fail Fast rule
 - Run Smoke tests 1st
 - Run faster tests 1st etc.

Test in a Shared Environment

- Tests should work whatever the environment
 - Should not require to run on John's machine
- Best solution is to provide appropriate shared test environments
 - To support integration testing
 - To support deployment / installation testing
 - To support system and smoke tests
 - To support Acceptance tests / User testing
 - To support performance, stability, stress testing etc.

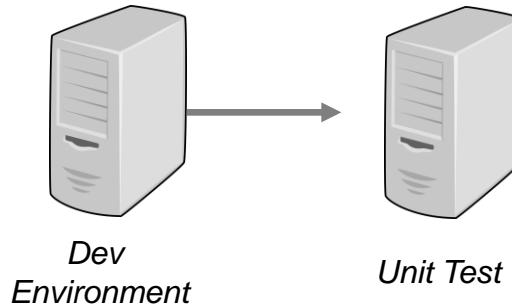
Different Test Environments



*Dev
Environment*

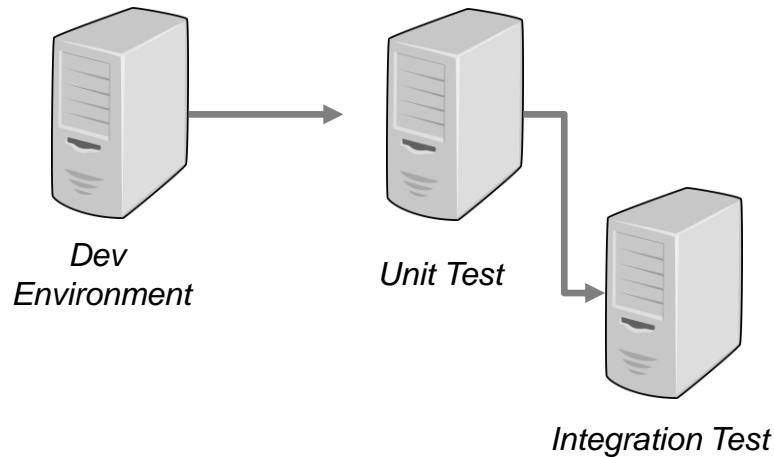
Write and Run Tests
Locally First

Different Test Environments



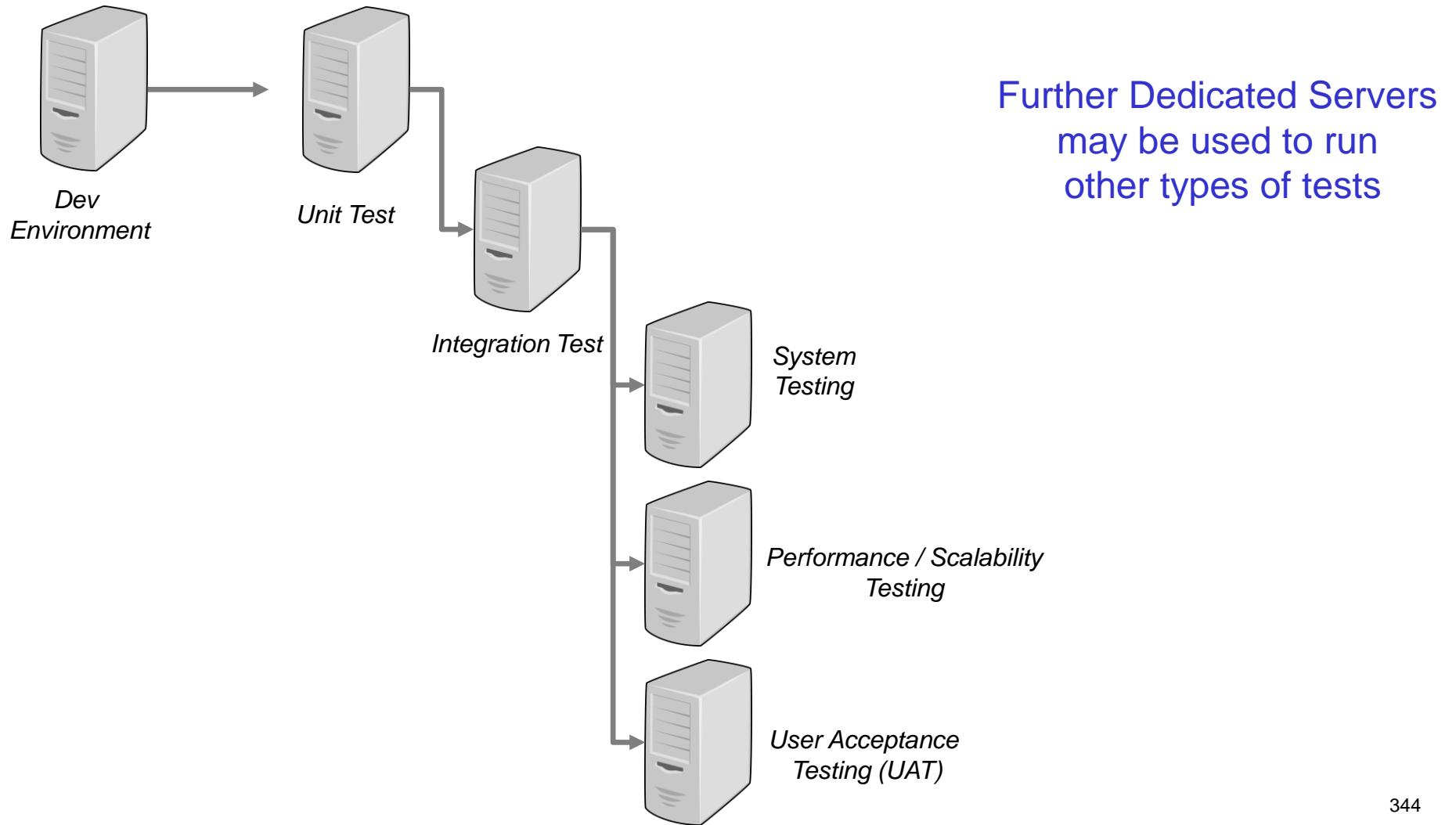
Run Unit Tests on
a CI Server

Different Test Environments

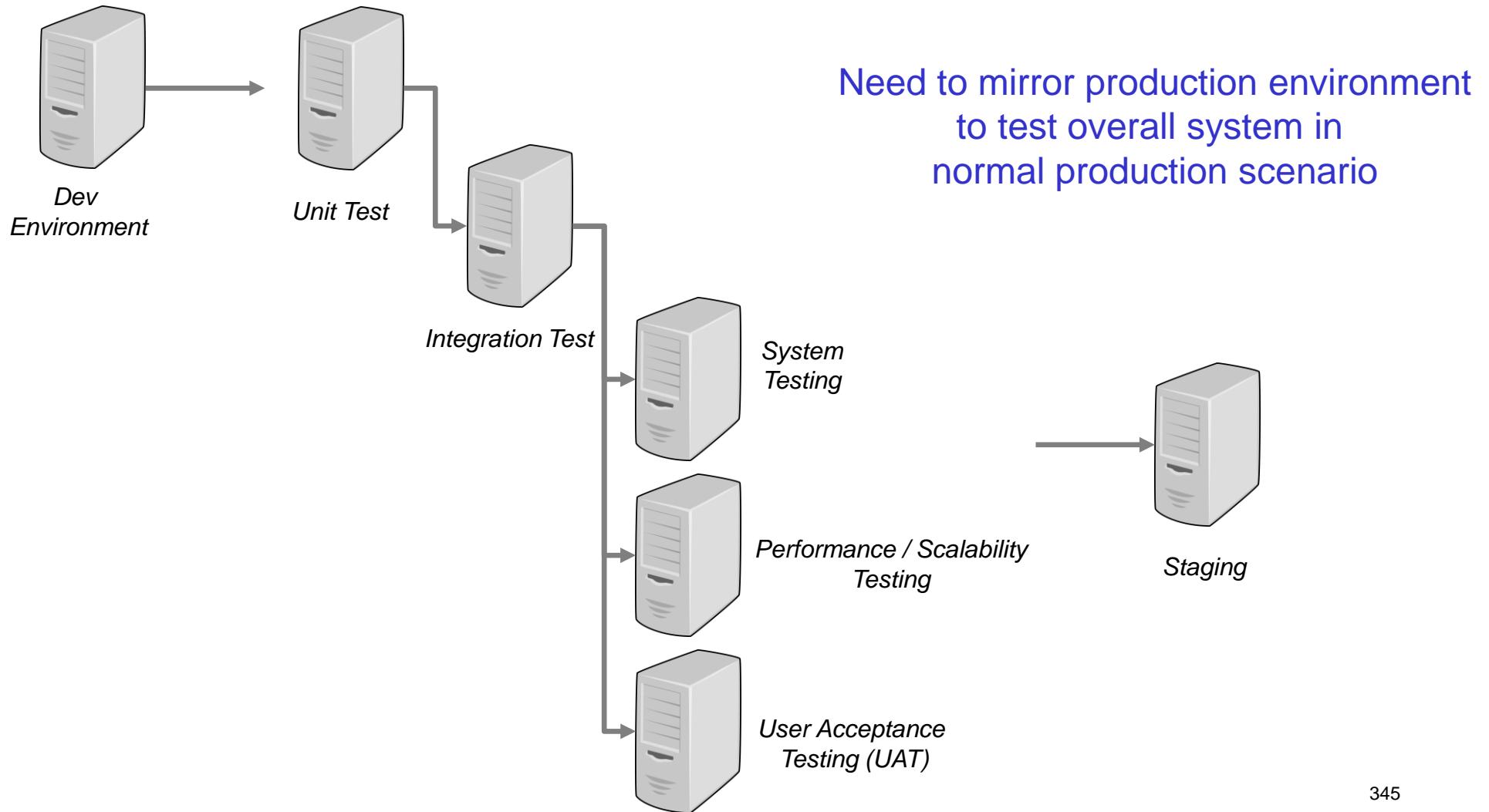


Run Integration Tests on
an Integration Server
design to run larger, longer running tests

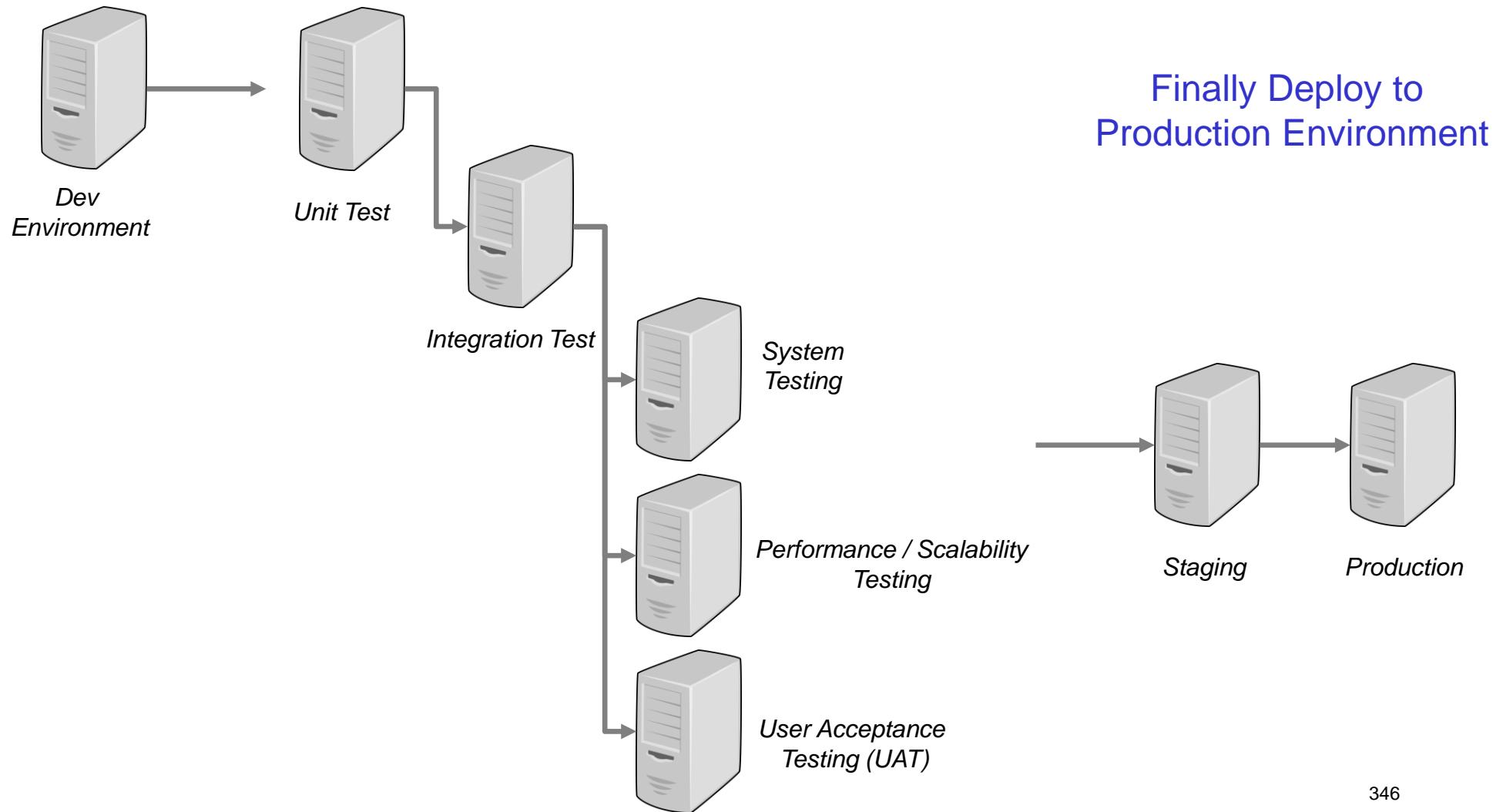
Different Test Environments



Different Test Environments



Production Environments



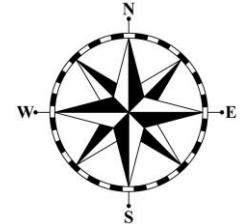
Writing Java Tests using JUnit 5



Toby Dussek

Informed Academy





Plan for Session

- Unit Testing - the very idea!
- JUnit 5.x Introduction
- Some terminology
- Setting up JUnit
- Conventions for tests
- Sample test
- Fixtures
- Nested tests, Test Names, Parameterized Tests
- Mocking and Mockito

JUnit 5.x



- What is JUnit
 - Java framework for unit testing
 - Available in other languages too
 - Freely available project
 - www.junit.org
- JUnit 5.0 on...
 - A major revision of JUnit framework
 - JUnit 5 = *JUnit Platform* + *JUnit Jupiter* + *JUnit Vintage*
 - Junit Platform runs tests on the JVM
 - Junit Jupiter provides model for writing tests
 - Junit Vintage provides a TestEngine for running JUnit 3 and JUnit 4 tests
 - requires Java 8 or higher

Some Terminology

- Test Class – a class with some unit tests in it
- Test Fixture – supporting operation for 1 or more tests
- Test (or Test Method) – a test implemented in a test Class
- Test Suite – a set of tests grouped together
- Test Harness / Runner – The tool that actually executes the tests

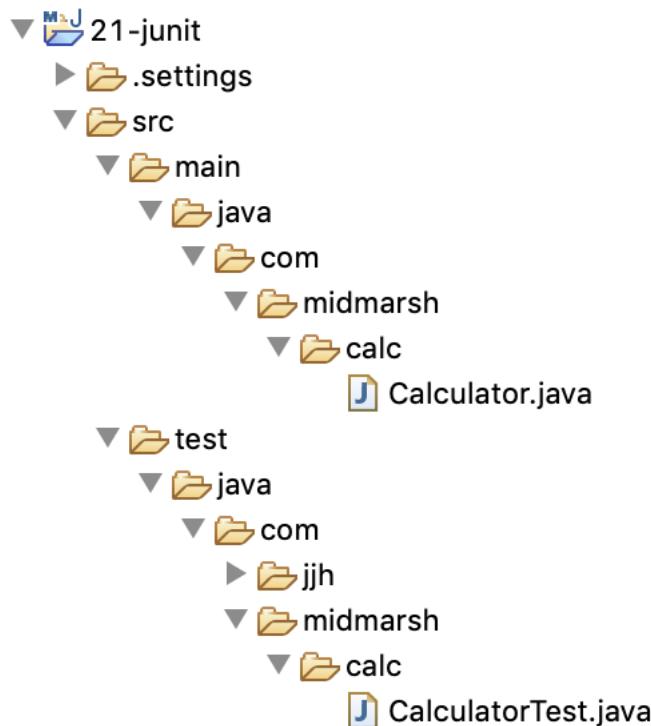
Using JUnit with Maven

- Need to add JUnit dependency to your POM

```
<dependencies>
  <dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-api</artifactId>
    <version>5.7.0</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-engine</artifactId>
    <version>5.7.0</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-params</artifactId>
    <version>5.7.0</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

Locating your tests

- Should have a test directory along side main



- May have src for JUnit tests
- May have resources for resources used by the tests

Conventions for Tests

- Many options available
- Choose which suits your project the best
- Help the IDE organise your test code:
 - Precede test name with Test: TestAccount
 - Suffix test name with Test: AccountTest
- Help tools to build your project:
 - Create additional source folders for test code

Sample Test

- Test class for simple Calculator
 - note naming convention used here

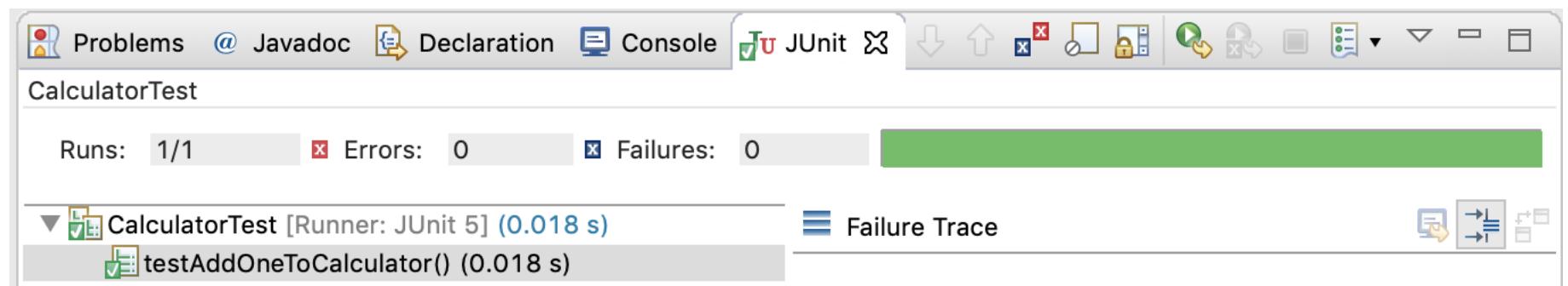
```
import static org.junit.jupiter.api.Assertions.assertEquals;
import org.junit.jupiter.api.Test;

class CalculatorTest {

    @Test
    void testAddOneToCalculator() {
        Calculator calc = new Calculator();
        calc.add(1);
        int result = calc.getTotal();
        assertEquals(1, result, "Expected result should be 1");
    }
}
```

Sample Test

- Can run from within an IDE



Validating Doubles

- Floating point numbers can be difficult to test
 - internal representation can result in numerous values after decimal point

```
double myPi = 22.0d / 7.0d;  
System.out.println(myPi);
```

3.142857142857143

- Can make testing hard
 - assertEquals takes a "fuzz factor,"
 - since doubles may not be exactly equal, fuzz factor lets you describe how close they have to be.

```
double myPi = 22.0d / 7.0d;  
assertEquals(3.14, myPi, 0.01);
```

Fixtures

- Behaviour to be run before and after
 - each test or
 - test classes
- Can be used for house keeping activities
 - set up object under test
 - reset object under test
 - release resources etc.
- Fixture methods indicated via annotations
 - @BeforeAll / @AfterAll run once per class (static methods)
 - @BeforeEach / @AfterEach run once per test

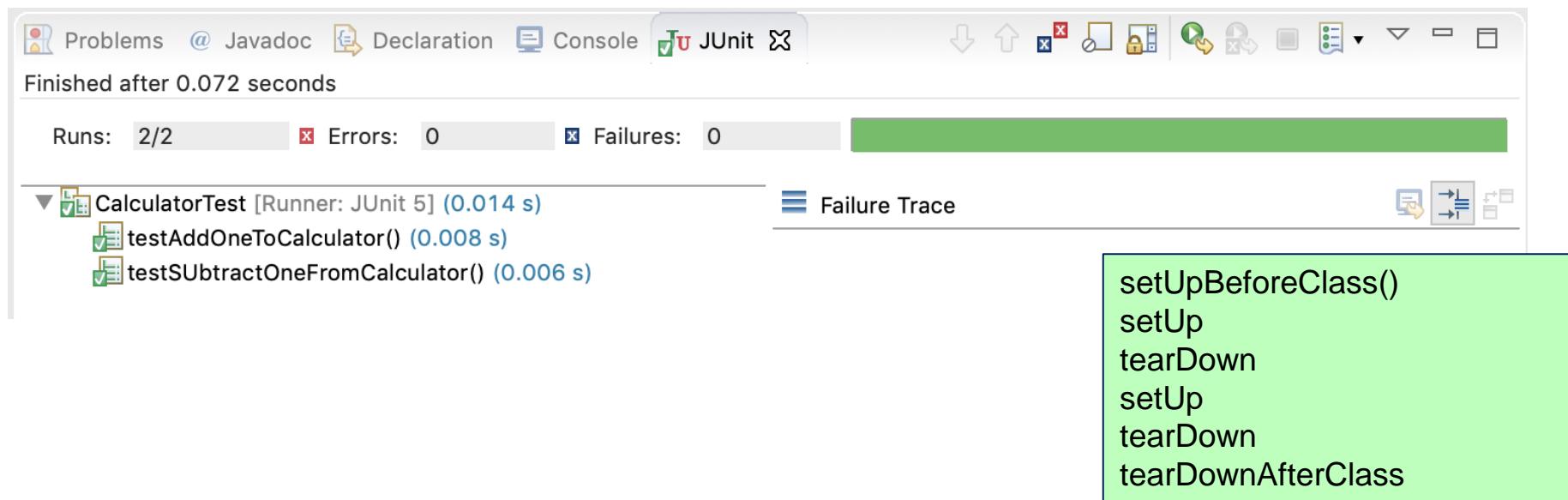
Fixtures

- Static methods for one time fixtures
- Instance methods for per test fixtures

```
class CalculatorTest {  
    @BeforeAll  
    static void setUpBeforeClass() throws Exception {  
        System.out.println("setUpBeforeClass()");  
    }  
    @AfterAll  
    static void tearDownAfterClass() throws Exception {  
        System.out.println("tearDownAfterClass()");  
    }  
    @BeforeEach  
    void setUp() throws Exception {  
        System.out.println("setUp");  
    }  
    @AfterEach  
    void tearDown() throws Exception {  
        System.out.println("tearDown");  
    }  
    @Test  
    void testAddOneToCalculator() {  
        Calculator calc = new Calculator();  
        calc.add(1);  
        int result = calc.getTotal();  
        assertEquals(1, result, "Expected 1");  
    }  
    @Test  
    void testSubtractOneFromCalculator() {  
        Calculator calc = new Calculator();  
        calc.subtract(1);  
        int result = calc.getTotal();  
        assertEquals(-1, result, "Expected -1");  
    }}}
```

Fixtures

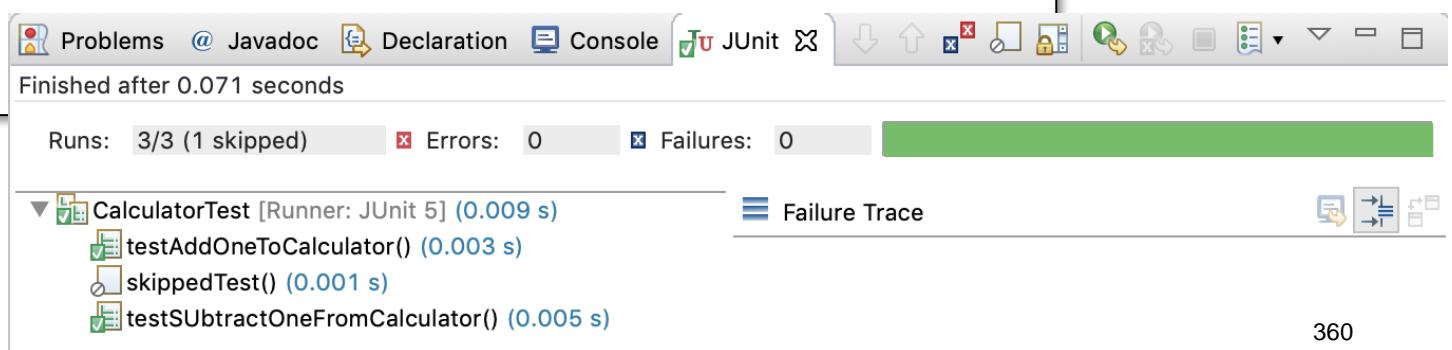
- ❑ Before and after class run once



Skipping Test

- Can disable a test
 - test will be ignored but reported on

```
import static org.junit.jupiter.api.Assertions.*;  
import org.junit.jupiter.api.*;  
class CalculatorTest {  
    // ... fixtures and other tests left out for space  
  
    @Test  
    @Disabled("for demonstration purposes")  
    void skippedTest() {  
        // not executed  
    }  
}
```



Conditional Tests

- Can run tests based on OS or Env Variables

```
import static org.junit.jupiter.api.Assertions.*;  
import org.junit.jupiter.api.*;  
class CalculatorTest {  
    // ... fixtures and other tests left out for space  
  
    @Test  
    @EnabledIfEnvironmentVariable(named = "ENV", matches = "staging-server")  
    void onlyOnStagingServer() {  
        // not executed  
    }  
  
    @Test  
    @DisabledIfEnvironmentVariable(named = "ENV", matches = ".*development.*")  
    void notOnDeveloperWorkstation() {  
        // ...  
    }  
}
```

Test Order

- Can control the order of tests

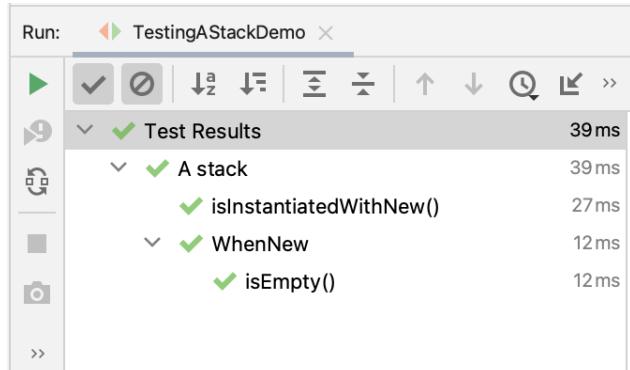
```
import org.junit.jupiter.api.MethodOrderer.OrderAnnotation;
import org.junit.jupiter.api.*;  
  
@TestMethodOrder(OrderAnnotation.class)
class OrderedTestsDemo {
    @Test
    @Order(1)
    void nullValues() {
        // perform assertions against null values
    }

    @Test
    @Order(2)
    void emptyValues() {
        // perform assertions against empty values
    }

    @Test
    @Order(3)
    void validValues() {
        // perform assertions against valid values
    }
}
```

Nested Tests

- Can nest tests
 - to allow related tests to be grouped together
- Use the
 - @Nested annotation for nested test class
- Can test to any level



```
@DisplayName("A stack")
class TestingAStackDemo {
    Stack<Object> stack;

    @Test
    void isInstantiatedWithNew() {
        new Stack<>();
    }

    @Nested
    class WhenNew {
        @BeforeEach
        void createNewStack() {
            stack = new Stack<>();
        }
        @Test
        void isEmpty() {
            assertTrue(stack.isEmpty());
        }
    }
}
```

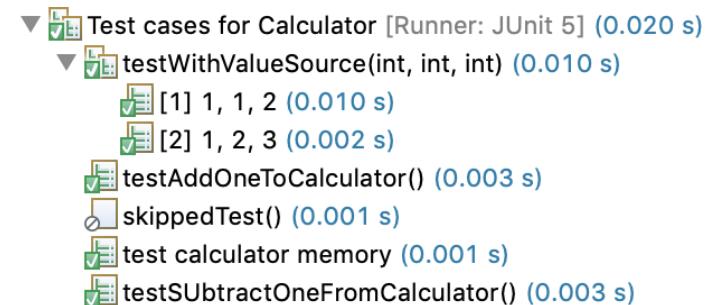
Parameterized Tests

```
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.*;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.Arguments;
import org.junit.jupiter.params.provider.MethodSource;

@DisplayName("Test cases for Calculator")
class CalculatorTest {
    // ... fixtures and other tests left out for space

    @ParameterizedTest
    @MethodSource("valuesProvider")
    void testWithValueSource(int x, int y, int total) {
        Calculator calc = new Calculator();
        calc.add(x);
        calc.add(y);
        int result = calc.getTotal();
        assertEquals(total, result, "result " + result + " should be " + total);
    }

    static Stream<Arguments> valuesProvider() {
        return Stream.of(Arguments.of(1, 1, 2), Arguments.of(1, 2, 3));
    }
}
```



Exception Testing

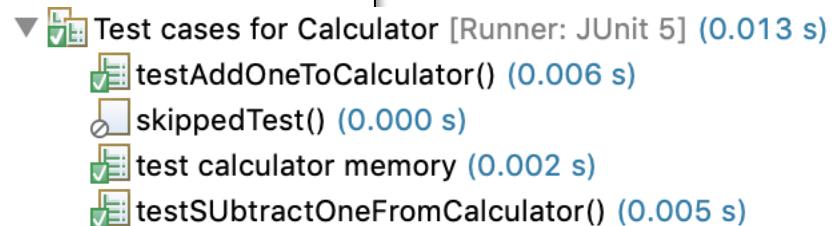
- Managed via assertThrows

```
@Test  
void shouldThrowException() {  
    Throwable exception = assertThrows(  
        UnsupportedOperationException.class, () -> {  
            throw new UnsupportedOperationException("Not supported");  
        });  
    assertEquals(exception.getMessage(), "Not supported");  
}  
  
@Test  
void assertThrowsException() {  
    String str = null;  
    assertThrows(IllegalArgumentException.class, () -> {  
        Integer.valueOf(str);  
    });  
}
```

Test Names

- Can give tests display names
 - containing space (may be more readable)

```
import static org.junit.jupiter.api.Assertions.*;  
import org.junit.jupiter.api.*;  
  
@DisplayName("Test cases for Calculator")  
class CalculatorTest {  
    // ... fixtures and other tests left out for space  
  
    @Test  
    @DisplayName("test calculator memory")  
    void testWithDisplayNameContainingSpaces() {  
        Calculator calc = new Calculator();  
        calc.add(1);  
        calc.addToMemory();  
        int result = calc.getMemory();  
        assertEquals(1, result, "Expected result 1");  
    }  
}
```

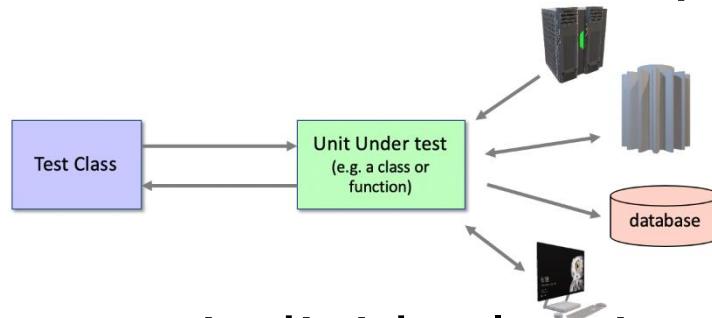


JUnit 4.x v JUnit 5.x

JUnit 4	JUnit Jupiter
@org.junit.Test	@org.junit.jupiter.api.Test <i>(No expected and timeout attributes)</i>
@BeforeClass	@BeforeEach
@AfterClass	@AfterEach
@Before	@BeforeEach
@After	@AfterEach
@Ignore	@Disabled

What is Mocking?

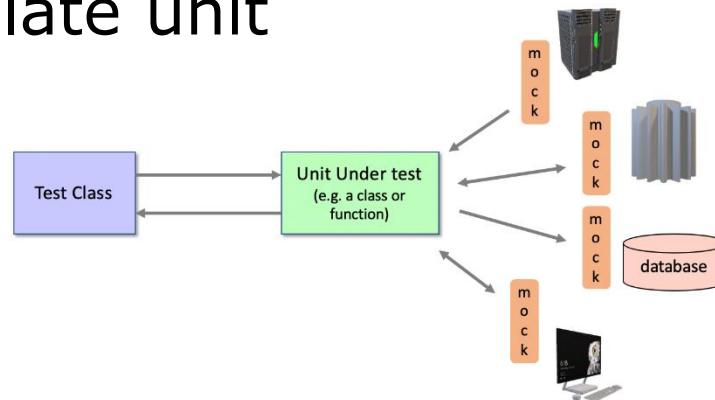
- Most systems interact with other systems



- But want to test an individual unit



- So need to isolate unit



Why Mock?

- Testing in isolation is easier
- The real thing is not available
- Real elements can be time consuming
- The real thing takes time to set up
- Difficult to emulate certain situations
- We want repeatable tests
- The Real System is not reliable enough
- The Real System may not allow tests to be repeated

Mocking In General

- A Mock is something that
 - Possess the same *interface* as the real thing
 - Define behaviour that in some way represents / mimics real exemplar behaviour
 - but typically in very controlled ways
- Different types of Mock
 - Test stubs
 - Fakes
 - Autogenerated Test Mocks
 - Test Mock Spies



Mockito

- A Test spy style Mocking framework
 - See <http://code.google.com/p/mockito/>
- Add Mockito Jar to test classpath
 - Or use maven dependency

```
<dependency>
  <groupId>org.mockito</groupId>
  <artifactId>mockito-junit-jupiter</artifactId>
  <version>3.0.0</version>
  <scope>test</scope>
</dependency>
```

- Mockito can be used with JUnit 4 and 5
 - Mockito uses Hamcrest
 - Mocks concrete classes as well as interfaces

Mockito

- Slim API
 - Only one kind of mock / only one way of creating mocks
 - Little annotation syntax sugar - @Mock
- General approach
 - Define expected behaviours before execution
 - Verify interactions after execution
 - E.g. exact-number-of-times and at-least-once verification
- Can use the Mockito.mock factory method
 - Processor processor = Mockito.mock(*Processor.class*);
 - mock all public methods in the Processor interface (or class)
 - will also provide default return values

Defining Mock Behaviour

- Referred to as Stubbing in Mockito terminology
- This is defining canned responses to Mock object methods
- Examples:
 - when(list.get(0)).thenReturn("Hello");
 - when(mockedMap.get("key")).thenReturn("hello");
 - when (methodCalled).thenThrow(anException);
 - doThrow(new
IllegalStateException("Illegal")).when(dao).get(1L);

Returning Values

- Can chain a series of return values
 - using series of .thenReturn

```
Mockito.when(processor.hasNext())
        .thenReturn(true)
        .thenReturn(true)
        .thenReturn(false);
```

- Can return a value for specified parameters
 - when(mockedList.get(anyInt())).thenReturn("element");
- Can also specify values
 - when(service.get(eq(1), anyFloat())).thenReturn("x");

Verifying Behaviour

- verify helps to verify that methods were called
 - “n” times, at least once or were never called etc.
- Check to see that a method was called
 - `verify(mockedList).get(anyInt());`
 - `verify(mockedList).add("one");`
 - `verify(mockedList).times(n)`
 - `verify(mockedList).atLeastOnce()`
 - `verify(mockedList).atLeast(n) / verify(mockedList).atMost(n)`
 - `verifyZeroInteractions(mock1, mock2, ...);`

Writing a Test using Mockito (1)

- Import JUnit and Mockito classes

```
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.mockito.Mockito;
```

- Write a setup Fixture

- to initialise the Mock

```
@BeforeEach
public void setUp() throws Exception {
    // Create mock object based on interface
    processor = Mockito.mock(Processor.class);
    // Set up what should happen when called
    Mockito.when(processor.next())
        .thenReturn(new Person("John", 47, "ABC123"))
        .thenReturn(new Person("Denise", 44, "XYZ987"));
    Mockito.when(processor.hasNext())
        .thenReturn(true)
        .thenReturn(true)
        .thenReturn(false);
    payroll = new Payroll(processor);
}
```

Writing a Test using Mockito (2)

□ Write test

- Invokes behaviour on unit under test
- Validates the results
- Confirms that the embedded processor was called the correct number of times

```
@Test
public void testPayrollGeneratesCorrectNames() {
    List<String> names = payroll.getPeopleToPay();
    assertNotNull(names);
    assertEquals("John", names.get(0));
    assertEquals("Denise", names.get(1));
    // We verify our mock calls
    // i.e. that next was called twice
    Mockito.verify(processor, Mockito.times(2)).next();
}
```

Using annotations

- Mockito can create mocks
 - Using the @Mock annotation
- Need to use the
 - MockitoExtension
 - with @ExtendWith
 - For this to work

```
import org.junit.jupiter.api.extension.ExtendWith;
import org.mockito.*;
import org.mockito.junit.jupiter.MockitoExtension;
// ... other imports omitted for space

@ExtendWith(MockitoExtension.class)
public class PayrollTest2 {
    @Mock
    private Processor processor;
    private Payroll payroll;

    @BeforeEach
    public void setUp() {
        System.out.println("setUp");
        System.out.println("processor: " + processor);
        // Set up what should happen
        Mockito.when(processor.next())
            .thenReturn(new Person("John", 47, "ABC123"))
            .thenReturn(new Person("Denise", 44, "XYZ987"));
        Mockito.when(processor.hasNext())
            .thenReturn(true)
            .thenReturn(true)
            .thenReturn(false);
        payroll = new Payroll(processor);
        System.out.println("payroll: " + payroll);
    }

    @Test
    public void testCanIterateOverPeople() {
        assertNotNull(processor);
        List<String> names = payroll.getPeopleToPay();
        assertNotNull(names);
        assertEquals("John", names.get(0));
        assertEquals("Denise", names.get(1));
    }
}
```