



Java

Fundamentals

Introduction to the course



Introduction to the course

Objectives

The aims and objectives of the course

Contents

Course administration

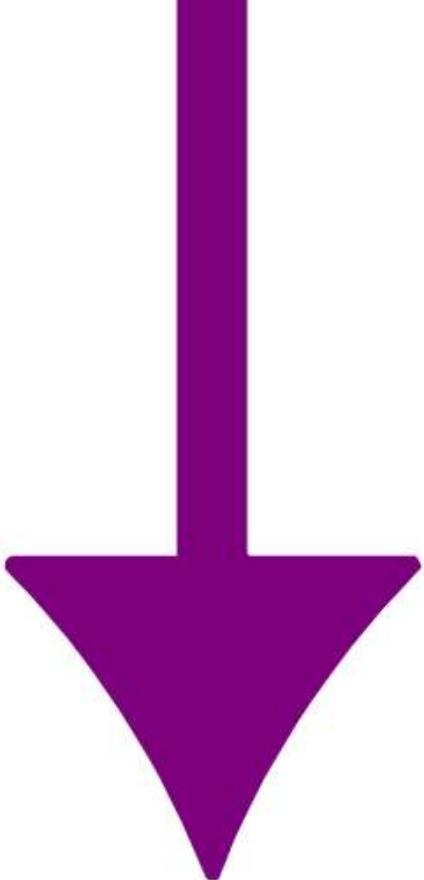
Objectives and assumptions

Introductions

Questions

Exercise

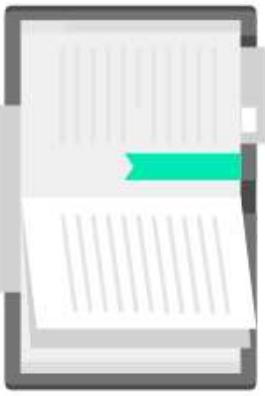
Explore the environment



QA Administration

- Front door security
- Name card
- Chairs
- Fire exits
- Toilets
- Coffee Room
- Taxis
- Trains / coaches
- Hotels
- First aid
- Timing
- Breaks
- Lunch
- Downloads and viruses
- Admin support

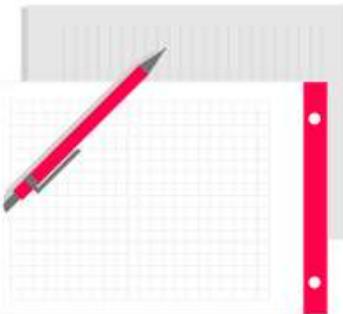
QA Course delivery



Learner guide



Exercise
guide



Questions
and exercises

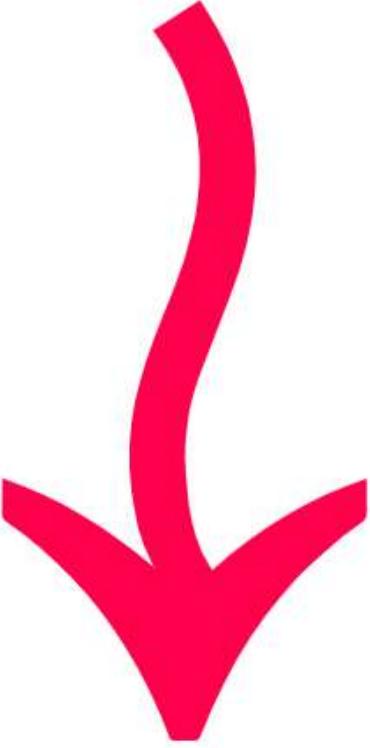


Practical
sessions

Hear and forget. See and remember. Do and understand.

Course outline

- Introduction to Java
- Primitives
- Control flow
- Introduction to objects
- Inheritance
- Interfaces
- Collections
- Exception handling
- File handling
- String manipulation and regular expressions
- Introduction to Lambda and streams
- JAR files and libraries
- Documentation
- Introduction to modules



QA

COURSE OBJECTIVES

By the end of this course you should be able to:

- Develop Java code using the IntelliJ IDE
- Write programs and run them from both the command line and the IDE
- Understand the basics of Java programming:
 - Creating objects and methods
 - Control flow within methods
 - Manipulating data
 - Access control
- Extend a superclass to create a subclass
- Use the main collection types
- Implement exception handling
- Work with simple files
- Understand the basics of Java functional programming
- Package up Java files for distribution

Assumptions

- This course assumes no previous Java knowledge
- The main requirement is that you're keen to program in Java!





Introductions

- By what name do you prefer to be called?
- Where are you based (work and home)?
- What programming experience might you have had?
- Have you done any Java before?
- What do you want to get out of this course?

Questions

Golden rule:

- 'There is no such thing as a stupid question'

First amendment to the golden rule:

- '...even when asked by an instructor'

Corollary to the golden rule:

- 'A question never resides in a single mind'



Course practical environment

In the course labs we will be using:

- Java 17
- IntelliJ IDEA
- Windows operating system
- A folder has been created for your code
C:\JAVA\JavaLabs\Labs

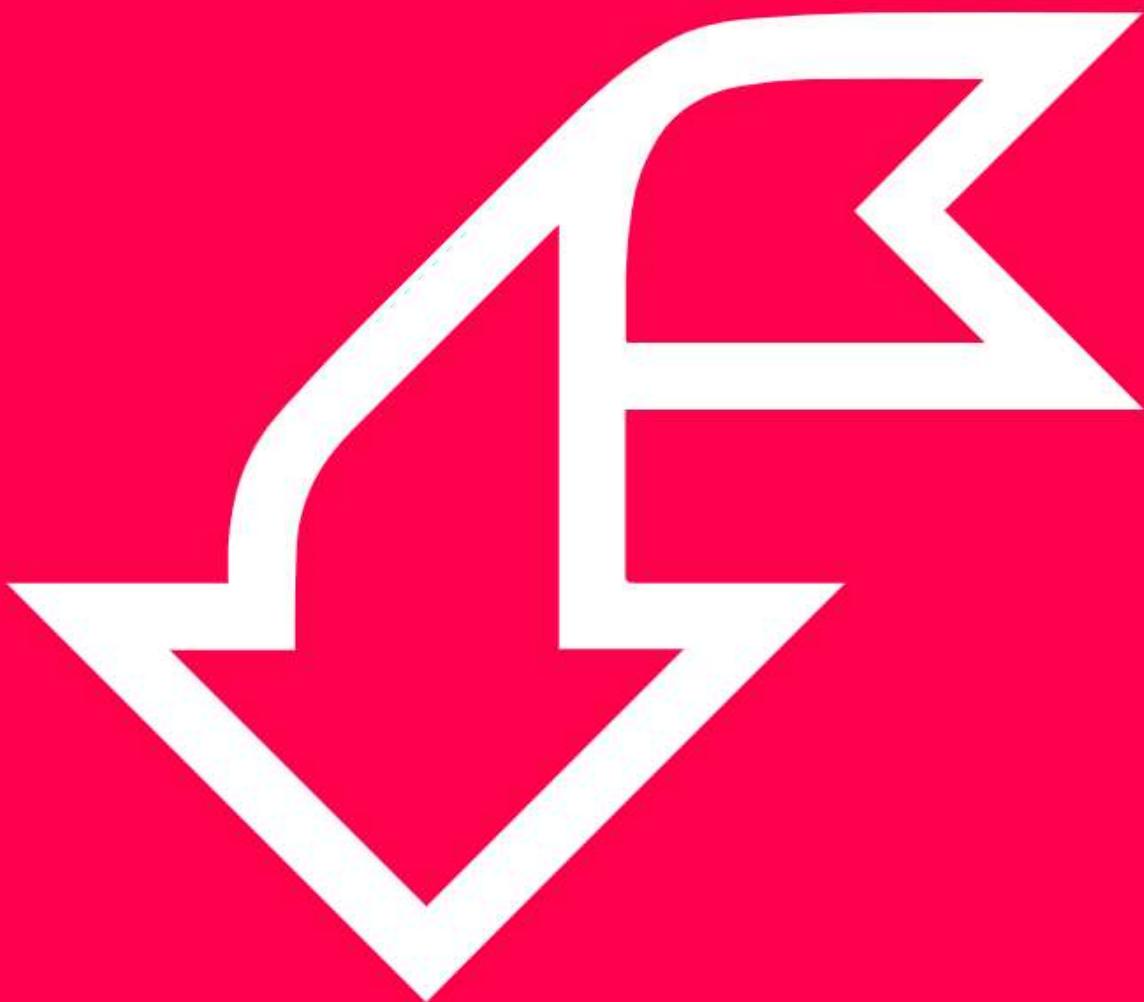
We will be discussing the Java Programming Language and IntelliJ in the next session, but first we're going to explore the environment and take an initial look at IntelliJ.



Lab 0 Getting started

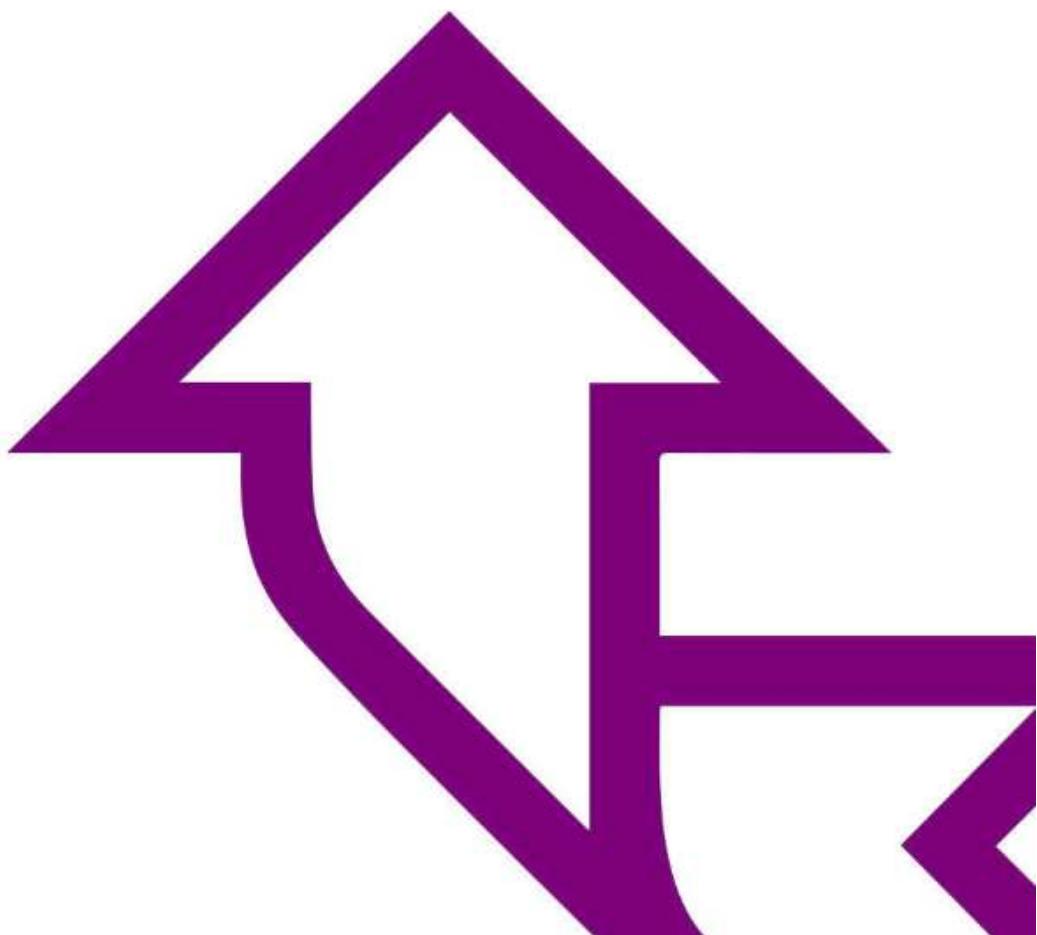
Explore the environment

- Check Java version
- Meet IntelliJ





Introduction to Java Programming



OUTLINE

Introduction to programming and Java

- What is a program?
- What is Java?
- Java design criteria
 - How Java works – the JVM
 - JRE and JDK
 - IDEs
 - IntelliJ – projects, adding items, editing code, running

Anatomy of a Java program

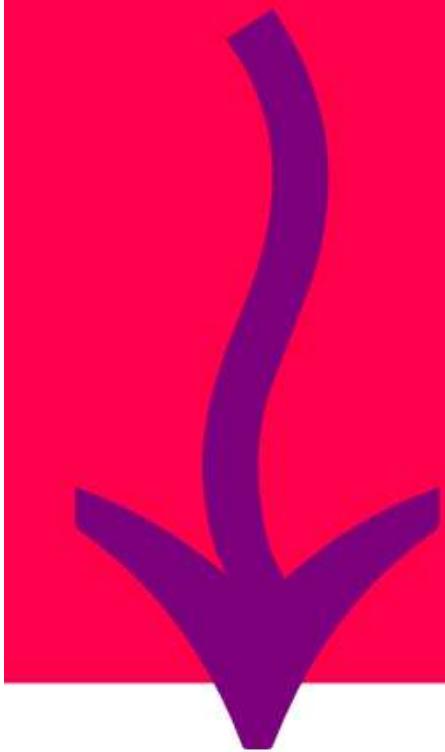
- Classes
- The main method and parameters
- Comments
- Output
- Packages

Running our program

OBJECTIVES

By the end of this session you should be able to:

- describe what a program is
- describe what Java is and understand the basics of how Java code is run
- write our first 'Hello World' program



vi

INTRODUCTION TO PROGRAMMING AND JAVA

WHAT IS A PROGRAM?

A program is a sequence of instructions to be carried out:

- Get a mug
- Find the coffee
- Open the coffee
- Put a teaspoon of coffee in the mug
- Add water
- Add milk (and sugar)

Programs:

- Describe the process of doing something
 - Doesn't need to be for a computer!
- Use a language both parties understand
- Contain and manipulate data

The instructions for a computer program are written in a high level language.

WHAT IS JAVA?

An object-oriented programming language

Popular due to the 'write once run anywhere' philosophy

- All code compiles down to bytecode which is run on a Java Virtual Machine (JVM)

- Operating system independent

- There are JVMs for just about any type of hardware

Huge number of libraries available

Well-supported:

- A lot of people and guides available to help

Portability:

- Take the exact same code and run on different servers

Java frameworks



JAVA DESIGN CRITERIA

Platform-independence

- Java source code (Xyz.java) compiles to processor-independent, platform agnostic ‘bytecodes’ (Xyz.class)
- Bytecodes then compile at runtime using platform specific JIT compilers, so same program runs on any platform supporting Java

Robust

- Fully object oriented, every line of code belongs to a class
- Strict type checking by compiler
- Built-in and enforced exception handling
- No pointers

Small and fast

- Each class is only loaded if needed
- Built-in multi-threading

Secure

- The runtime environment can restrict what a Java class can do



Q1 How does Java work?

Java source code is compiled into Java byte code

- This is the .class file

This is then run on the Java virtual machine (JVM)

The JVM translates between the byte code and the underlying operating system

This makes Java code portable

- The JVM implementation needs to be specific to the processor and operating system, but a compiled java class does not.



Q What is the JVM?

The Java Virtual Machine loads the required java classes that are present on the class path

- This can be the .class files you've written
- jar files you've included
- On the command line we include the parameter -cp
- In an IDE we add the jar files to the build path

The JVM verifies all the classes it loads

- Checks for any errors

It also holds the security manager

- This restricts access to the host machine or other files dependent on properties

Q A JRE vs. JDK

JRE: Java Runtime Environment

- What we need to run a Java program
- Installed on most machines

JDK: Java Development Kit

- The Java compiler and other tools we need to be able to create Java programs
- We need a JDK on any development system!

Q& What is an IDE?

Provide tools to make writing code easier:

- Auto-completion
- Error checking
- Debugging
- Code inspection
- Refactoring
- Auto generation of source code

Don't need to use one:

- Can just use a text editor and the command line
- Doing so is likely to be less productive

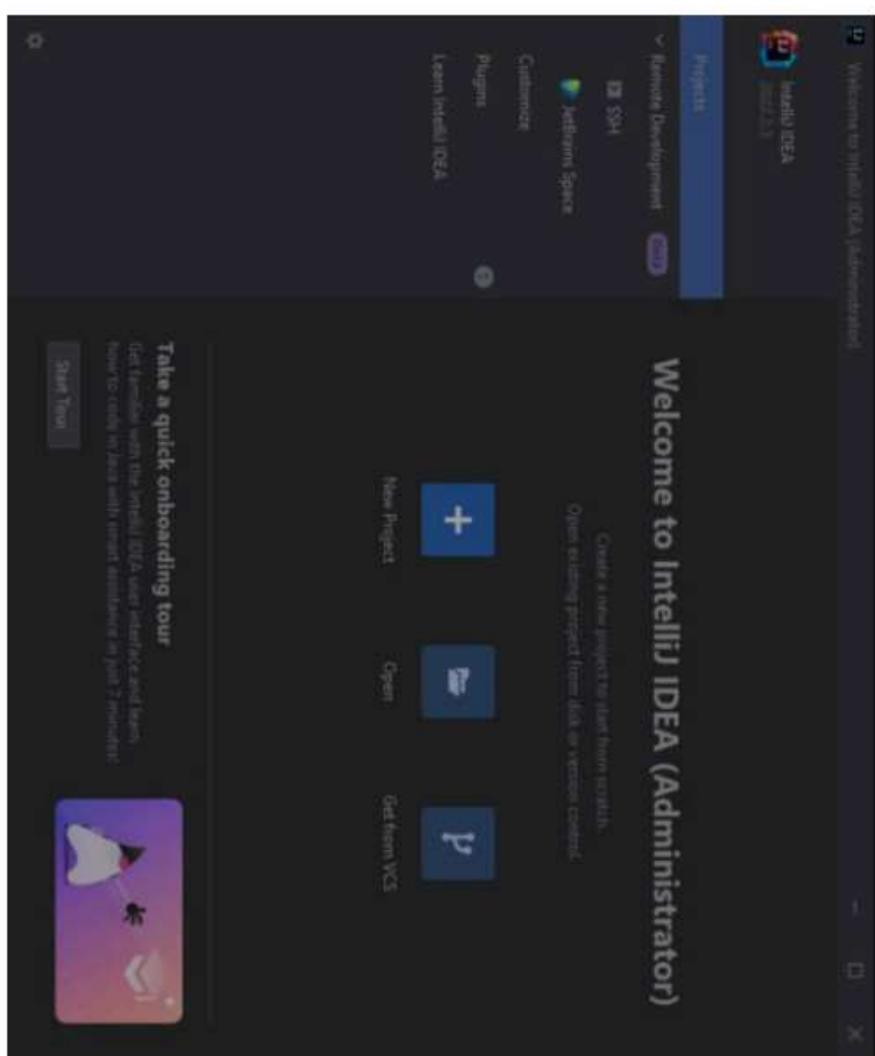
QA

What is IntelliJ?

IntelliJ is a leading Java IDE

Developed by JetBrains (formerly known as IntelliJ)

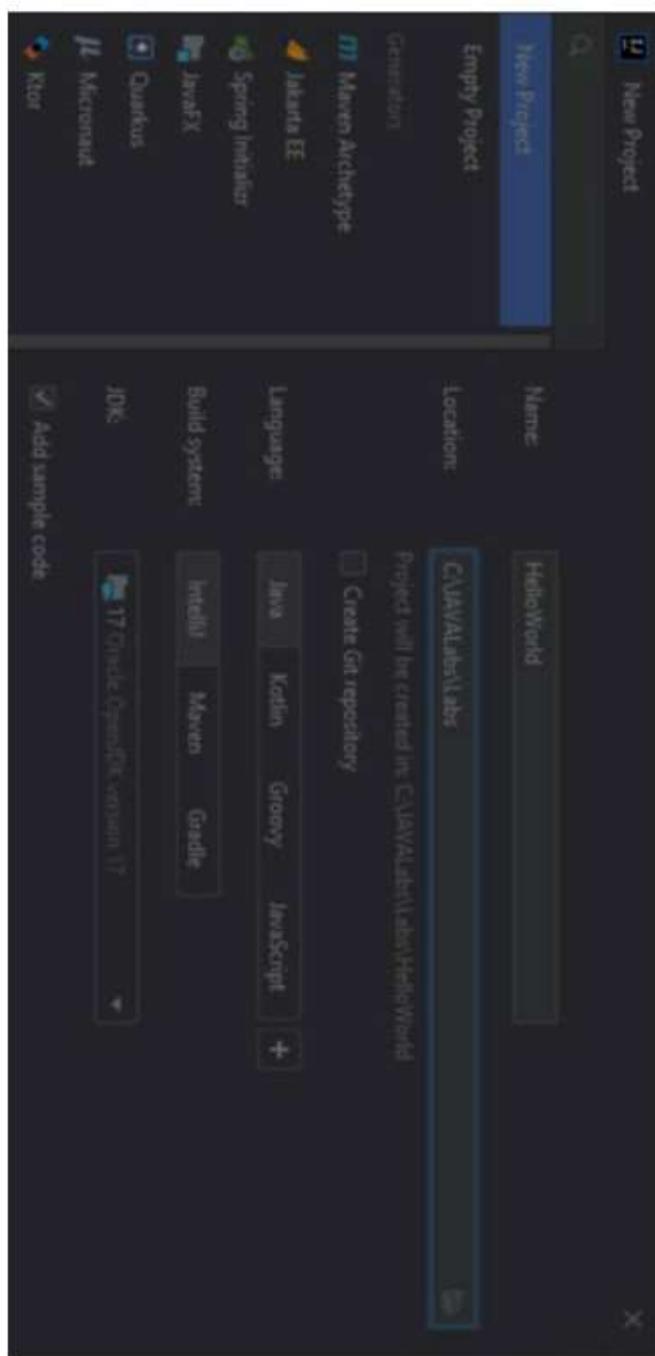
Available as a community edition and a propriety commercial edition



Projects

Each application is organised within a project

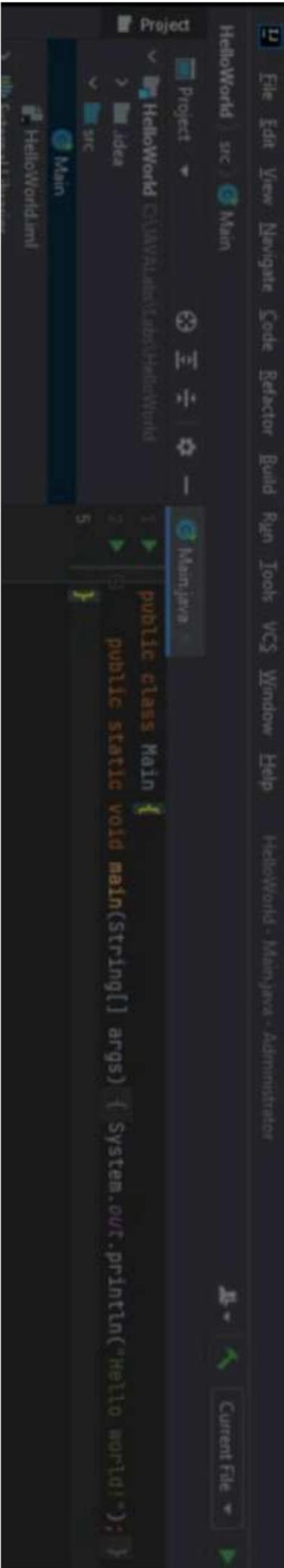
- Category chosen via dialog(s) using File > New
- Enter the Project Name and location
- Language **Java** and Build System **IntelliJ**
- Select the JDK
- Named sub-folder created in the specified location



Q4 Project tool window

Each project is shown in its own window – you can have several open

- Each project may have more than one IntelliJ module
- Each IntelliJ module has a src directory
 - There may be many Java packages in the src directory
 - Each package may have many classes or classes may be created directly in src
 - Tree View under the project name expands to show the packages (not shown here) and classes

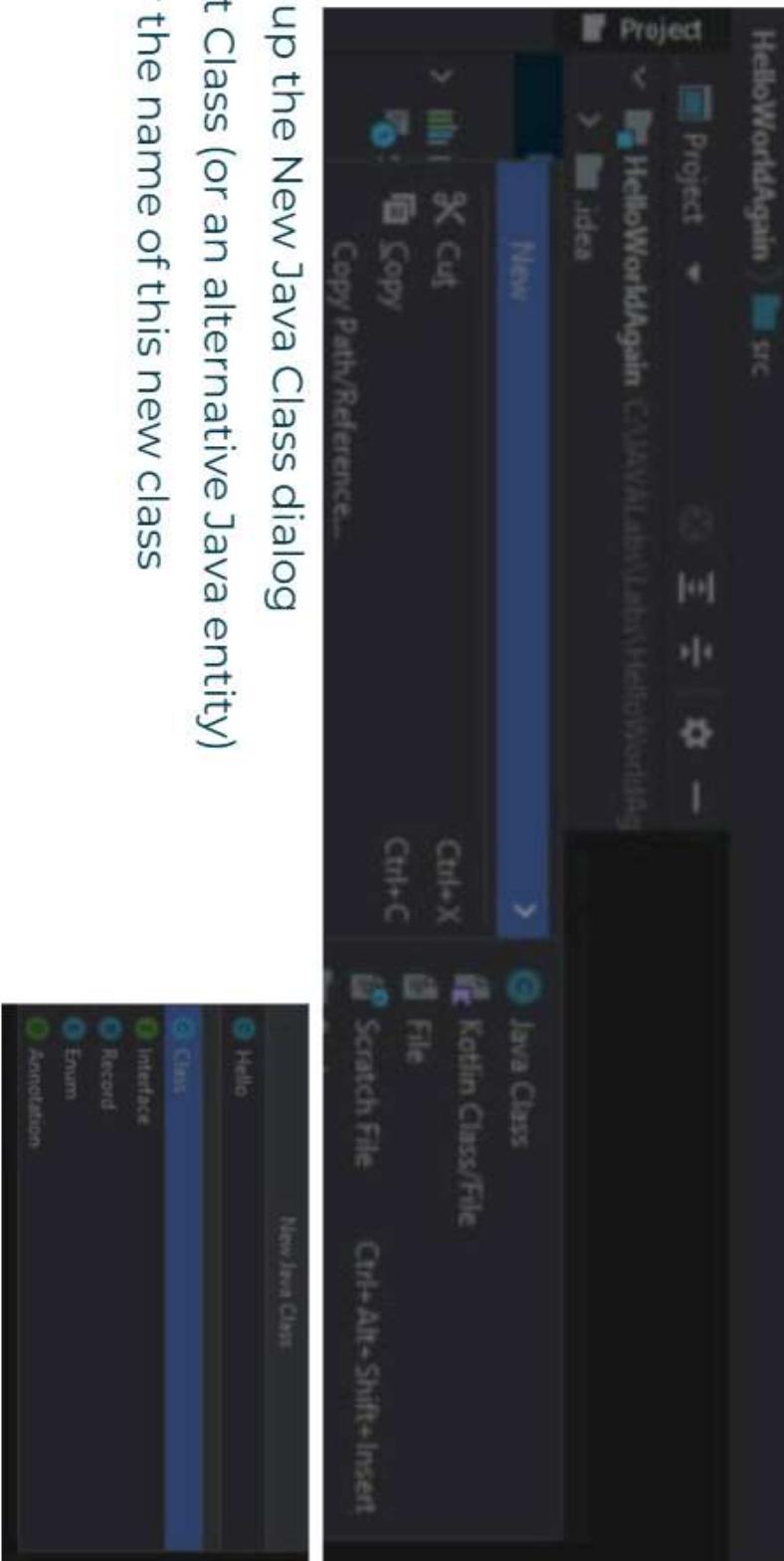


Qn Adding items to projects

Projects start empty

For new items, right-click src in the project to open the context menu

New | Java Class



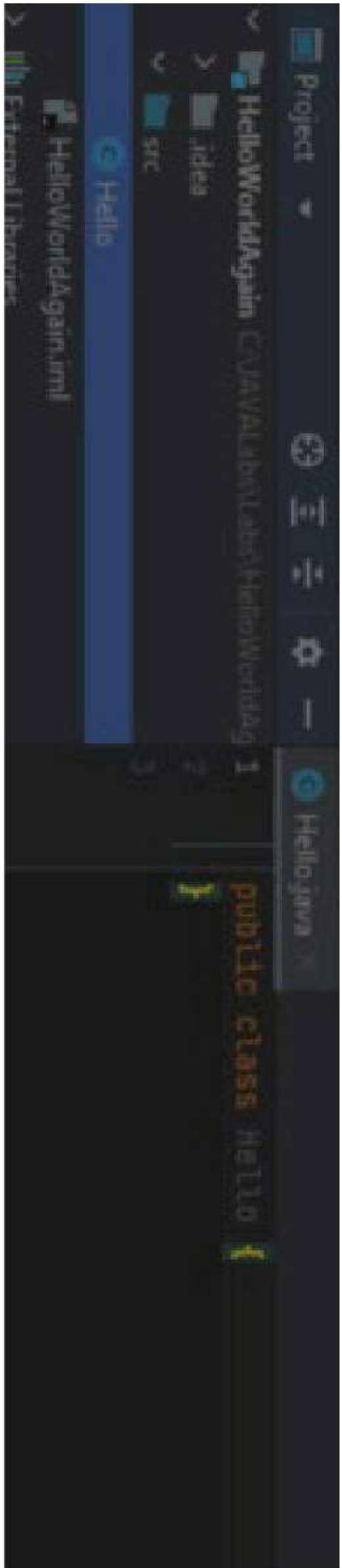
Opens up the New Java Class dialog

- Select Class (or an alternative Java entity)
- Enter the name of this new class

QA Class file

The Class source file is added to the project's tree structure

- Open in the main pane (by default)
- IntelliJ supplies an empty class for you

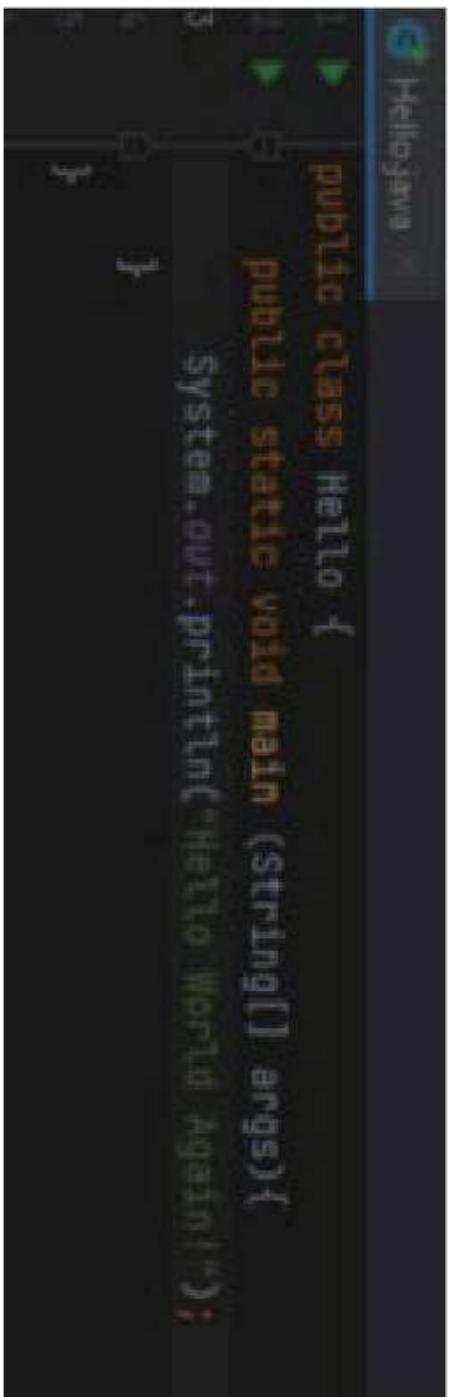


Hello.java file now available for editing

QA Getting started

All code is added via the editor pane

- Simply type java code
- Colour and font 'coded'



```
 1 package com.raywenderlich.android;
 2
 3 public class Hello {
 4     public static void main (String[] args){
 5         System.out.println("Hello World Again!");
 6     }
 7 }
```

The application is ready to run

ANATOMY OF A JAVA PROGRAM

Q&A Anatomy of a Java program – writing our first class

The JVM looks for a **main** method

- This is a specific method that sets up and runs the entire program
- Must be public: otherwise it cannot be seen from outside the class
- Must be static: otherwise we need to create an object
 - (Don't worry about this too much yet)
- Must have the parameters: String[] args: These are the arguments that can be passed to the program

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

Qn Anatomy of a Java program – classes

The first line of our program is the **class identifier**

- **public** means that other parts of a Java program can see this class
- Classes are normally declared **public** unless there is a very specific reason not to
- **class** is a keyword to tell Java that this is a class description
- **Hello** is the name of this class
 - any name appropriate to the application
 - starts with a capital letter by convention

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

Q& Anatomy of a Java program - the main method

- **public** means that this method can be seen outside the class
- **static** means we don't have to create an instance of the class before calling the method
- **void** is the return type – we're not returning a value so use the keyword void
- **main** is the method name – for a main method the name is important

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

Q1 Method parameters

The method parameters are what we may pass to the method for it to use

In the main method, this comes from either the command line when we run the program or in an IDE it comes from the run settings

String[] args – an array of String objects passed in, with the name ‘args’

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

Qn Anatomy of a Java program - comments

Comments are ignored by the Java compiler

They are there to let people reading the code know what is going on

Java has three types of comment

- *//* - single line comment
- */* ... */* - Multiline comments
- */** ... */* - Javadoc comments. Read by the Javadoc tool to create documentation

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

Qn Anatomy of a Java program - output

The final line prints out to the console

- System – refers to a Java class called **System**
- out – access the **out** PrintStream object in the **System** class
- println – call the **println** method which prints things to the console
- “Hello World” – input parameter for the **println()** method, this is the String we are printing out
- ; - Very important! Every Java statement ends with a semi-colon

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

QA Packages

Hierarchical grouping structure for organising Java files

- Each package contains a collection of classes
- Helps to avoid name clashes – class name is packageName.ClassName

Best practice is to create a class in a package

Java provides many different built in packages, including:

Function	Name	Example classes
Language	java.lang	Object, String, Thread, Runtime, System
Date & Time	java.time	LocalDate, LocalTime, LocalDateTime, Period
I/O	java.io	InputStream, OutputStream, File
Utilities	java.util	HashMap, TreeSet, ArrayList

Q4 Putting classes in packages

Create a package

- This is created as a folder in your project
- Right-click the src folder of your project (the root directory for all source code)

Select New | Package

Type the name for the new package(e.g. com.qa), then Enter

The package is added to the source tree

- To put a class into the package – right-click the package name and select New | Java Class
- In the class skeleton code in the Editor pane the package declaration will appear as the first line

package com.qa;

RUNNING OUR PROGRAM

Q4 Running a program from IntelliJ

IntelliJ checks the syntax for us continually

- Identifies problems as they occur

We have three options for compiling and running the program:

- From the editor, using the green arrow in the gutter (left hand margin)
- From the Run menu
- By selecting the green arrow to the top-right above the editor



QA Running from the Command Line

Java programs can be compiled and run on the command line

- Compile using javac
- Run using java

To compile go to the base directory for the program

- This is the src directory that IntelliJ created for us
- Give javac the full pathname

```
javac com\qa\Hello.java
```

- This will create a .class file in the same directory.

To run it we need to use the full package name of the file, but not the .class extension

```
java com.qa.Hello
```

Lab 1

Helloworld again

- Writing your own Helloworld Program
- Running it from IntelliJ and the command line

SUMMARY

- Introduction to programming and Java
- What is a Program?
- What is Java?
- Java Design Criteria
- How Java works – the JVM
- JRE & JDK
- IDEs
- IntelliJ – Projects, adding items, editing code, running

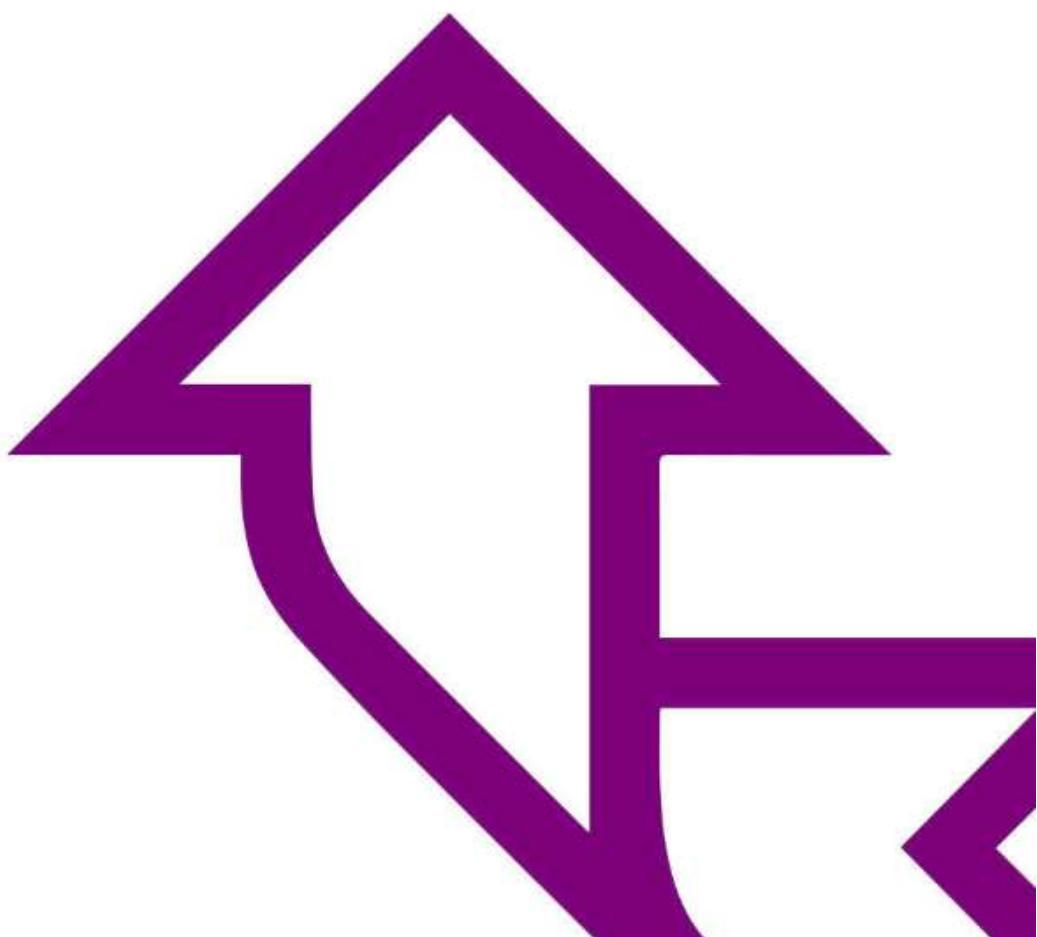
Anatomy of a Java program

- Classes
- The main method and parameters
- Comments
 - Output
 - Packages

Running our program



Basic Syntax



OUTLINE

Basic syntax

- Expressions, statements, and blocks
- Comments
- Variables

Primitive types

Simple maths and logical operators

Strings



OBJECTIVES

By the end of this session we should

- Understand and use primitives in Java
- Use some mathematical and logical statements in Java

BASIC SYNTAX

QA Expressions, statements, and blocks

Java is built using expressions, statements, and blocks of code

An expression

- Constructed from variables, operators, and method invocations
- Evaluates to a single value
- $x + y / z$

A statement

- A complete unit of execution
- Terminated with a semi colon
- `int apple = x + y / z;`

A block

- Code contained within { ... }

QA Comments

As seen earlier, there are three types of comments in Java

- //
 - Single line comment
 - Used to describe what is going on
- /* ... */
 - Multi-line comment
- /** ... /
 - Javadoc comment
 - Describes the code in a way that can be processed and picked up by the Javadoc compiler and turned into documentation

Why do we want comments?

QA Variables (fields)

Java variables

- Have a type, a name, and may have an initial value
- Can be assigned and reassigned different values

```
String hello = "hello";
int i = 5;
int age = 21;
double costOfApples = 3.20

i = 10;
age = 22;
```

When a variable is contained inside an object it is known as a field or attribute

QA Variable names

Case sensitive

- Hello is different from HELLO is different from hello

Must start with a letter, the \$ symbol or an underscore _

- Convention is to avoid \$ or _

Can include numbers

Can't include other punctuation (?!, or spaces)

Convention is to use camel case

- Small letter at the start of the variable name
- Capitalise the initials of other words

```
firstName  
numberOfApples  
weatherInBirmingham
```

Q4 Reserved words

These are keywords in the Java language

boolean
byte
char
double
float
int
long
short
void

abstract
final
native
private
protected
public
static
synchronized
transient
volatile

strictfp (1.2)
default (1.8)

false
null
true

break
case
catch
continue
default
do
else
finally
for
if

class
extends
implements
interface
throws

import
package

instanceof
new
super
this

try
throw
switch

const
goto

Reserved for
future use.

QA Quiz

Which of these names are legal?

- Alice
- alice
- ALICE
- 42Apples
- Number_Of_Apples
- \$percentage
- %ofCookiesForAlice
- Num Apples
- allTheApples_Are_mine
- someApples

Q& Quiz answers

Which of these names are legal?

- Alice
- alice
- ALICE
- 42Apples
- Number_Of_Apples
- \$percentage
- %ofCookiesForAlice
- Num Apples
- allTheApples_Are_mine
- someApples

Starts with a number

Can't use % in a variable name

Can't have spaces

PRIMITIVE TYPES

QA

THESE ARE EIGHT PRIMITIVE DATA TYPES



Name	Default value	Value range
byte	0	-128 : 127
short	0	-32,768 : 32767
int	0	- 2^{31} : $2^{31}-1$
long	0L	- 2^{63} : $2^{63}-1$
float	0.0f	
double	0.0d	
char	'\u0000'	'\uffff' (also values like 'a')
boolean	false	false / true

Q4 Declaring a variable

- Uses the following syntax:

```
type name = value;
```

```
int age = 5;  
boolean young = true;  
char a = 'a';  
double oldWAT = 17.5;
```

- You don't need to give it the value straight away, you can just name it initially

```
boolean young;  
char a;  
double oldWAT;
```

- You can also chain declarations together

```
int teamA, teamB, teamC;
```

QA Literals

Integer number

- Decimal, binary, octal or hexadecimal
- L or L specifies **long**

Floating point number

- Standard or scientific notation
- **double** type by default
- f or F specifies **float**

Boolean

- Can be **true** or **false**

Single character

- Character, escape, octal, Unicode
- Use **single quotes**

String of characters

- Implemented by the String class

Integer

0 1 42 -23795
02 077 0123 0b101
0x0 0x2a 0x1FF
3l 077L 0x1000L

Floating point

1.0 4.2 .47
1.22e19 4.61E-9
6.2f 6.21F

Boolean true false

Character

'a' '\n' '\t' '\077'
\u006F'

String

"Hello, world\n"

Qn Changing a variables value

You can assign a variable a value using the equals sign

This will change any previous value referred to by that variable

```
int age = 5;  
// the value of age is currently 5  
  
age = 16;  
//Changed the value to 16  
//5 is forgotten.
```

Variables must be assigned a value before they are used!

- The compiler will give an error if you use a variable that has not been given a value

QA

Quiz time!

Find the deliberate syntactic mistakes

Note: Assume errors are fixed as you go

```
byte sizeof = 200;  
short num = 43;  
short hello num;  
int big = sizeof * sizeof * sizeof;  
long bigger = big + big + big; / ouch /  
double old = 78.0;  
  
double new = 0.1;  
boolean consequence = true;  
boolean a, b; c;  
boolean max = big > bigger;  
char maine = "american state";  
char ming = 'd';
```

QA Answers

byte is 8-bit signed integer

range is -128 to +127

```
byte sizeof = 200;
```

short hello mum;

Need a token between
hello and mum

```
long bigger = big + big + big; / ouch /
```

Badly formed comment
- use /* */ or //

```
double new = 0.1;
```

new is a reserved word -
illegal as a variable name

```
boolean a , b ; c ;
```

'c;' is not a
statement

```
char maine = "american state";
```

maine is a 'char'

SIMPLE MATHS AND LOGICAL OPERATORS

Q1 Simple maths

Storing numbers isn't very useful if we can't do anything with them!

There are a number of simple built-in operators we can use

Addition	<code>int x = 1 + 5;</code>
Subtraction	<code>int x = 1 - 5;</code>
Multiplication	<code>int x = 1 * 5;</code>
Division	<code>int x = 1 / 5;</code>
Modulo	<code>int x = 1 % 5;</code>

More complex methods are available in the `java.lang.Math` class.

Q4 Compound assignment

These are combinations of an arithmetic operator with an assignment

x += 5	is the equivalent of	x = x + 5
x -= 5		x = x - 5
x *= 5		x = x * 5
x /= 5		x = x / 5

There are also some shortcuts that were borrowed from C

x++	x = x + 1
x--	x = x - 1

The position of the symbol affects the associativity

```
int var1 = 3, var2 = 0;  
var2 = ++var1;           // both now 4  
var2 = var1++;          // var1 = 5, var2 = 4
```

QA Comparisons

All return a true or a false value based on the numbers given

x > y	X is greater than Y
x < y	X is less than Y
x >= y	X is greater than or equal to Y
x <= y	X is less than or equal to Y
x != y	X is not equal to Y
x == y	X is equal to Y

Make sure you use two equals signs when comparing x equal to y
- otherwise it's an assignment statement!

QA Order of operations

If you don't use brackets in an arithmetic statement the order may appear ambiguous

- BODMAS used for order of precedence

Brackets

Orders/Indices (powers and square roots)

Division and Multiplication

Addition and Subtraction

Best practice is to use brackets to avoid ambiguity

```
int a = 1 + (5 * 2)
```

Q4 Logical operators

Return a Boolean value

<code>&&</code>	And (with short circuiting)
<code>&</code>	And (without)
<code> </code>	Or (with short circuiting)
<code>-</code>	Or (without)
<code>!</code>	Not
<code>^</code>	Xor

When using `&&` or `||` this short circuits the expression

- It doesn't evaluate more than is required

`(age >= 18) && (val == 7)`

`false && ...`
`true || ...`

Q4 Operator precedence

Order	Operators	Comments
1	. [] (param)	Postfix
2	++ -- ! ~ instanceof	Unary
3	new (type) expr	Creation and case
4	* / %	Multiply and Divide
5	+ -	Add and Subtract
6	<< >> >>>	Shift
7	< > <= >=	Relational
8	== !=	Equality
9	&	Bitwise AND
10	^	Bitwise exclusive OR
11		Bitwise inclusive OR
12	&&	Logical AND
13		Logical OR
14	?:	Conditional
15	= op=	Assignment

vi

STRINGS

Qn Strings

Strings are often thought of as very similar to primitive types

- Unlike primitives they are declared using a capital letter for String
- This is because they are objects
- But they can be assigned values in the same way as primitives

```
String hello = "Hello World";
```

- Strings can be concatenated using the + symbol

```
String greeting = "Hello";
String name = "Alice";
String sayHi = greeting + " " + name;
```

Qn Combining strings and other variables

You can combine Strings with other variables

- It converts the other variable into a String

```
String ageMsg = "My Age is " + 21;
```

When comparing Strings values we don't use ==

- As String is not a primitive type, the == symbol checks that the two Strings are references to the same place in memory – NOT if the contents are the same
- Use the **.equals** or **.equalsIgnoreCase** methods

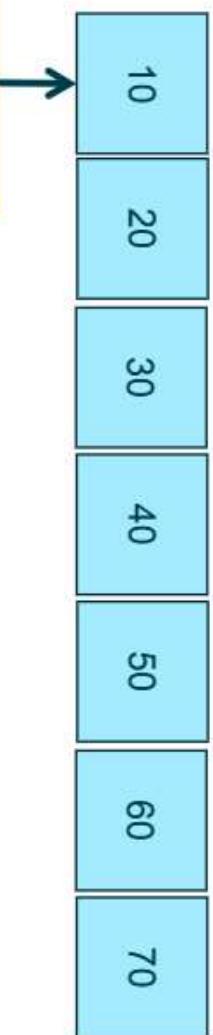
```
//will return a true or false  
boolean messagesMatch = ageMsg.equals(otherMsg);
```

Qn Arrays

An array allows us to store things as a basic list

- An array can have elements of any data type
- We declare it using square brackets next to the type

```
int[] arr = new int[7];
int[] arr2 = {10,20,30,40,50,60,70};
main(String[] args)
```



To access the items in the array we use its index:

```
int a = arr2[3]; // a = 40
arr2[0] = 100; //changes the value at position 0
```

Programming time

- Create different types of variables
- Use arithmetic and logical operations on them
- Print out the values to the screen

SUMMARY

Basic syntax

- Expressions, Statements and Blocks
- Comments
- Variables

Primitive types

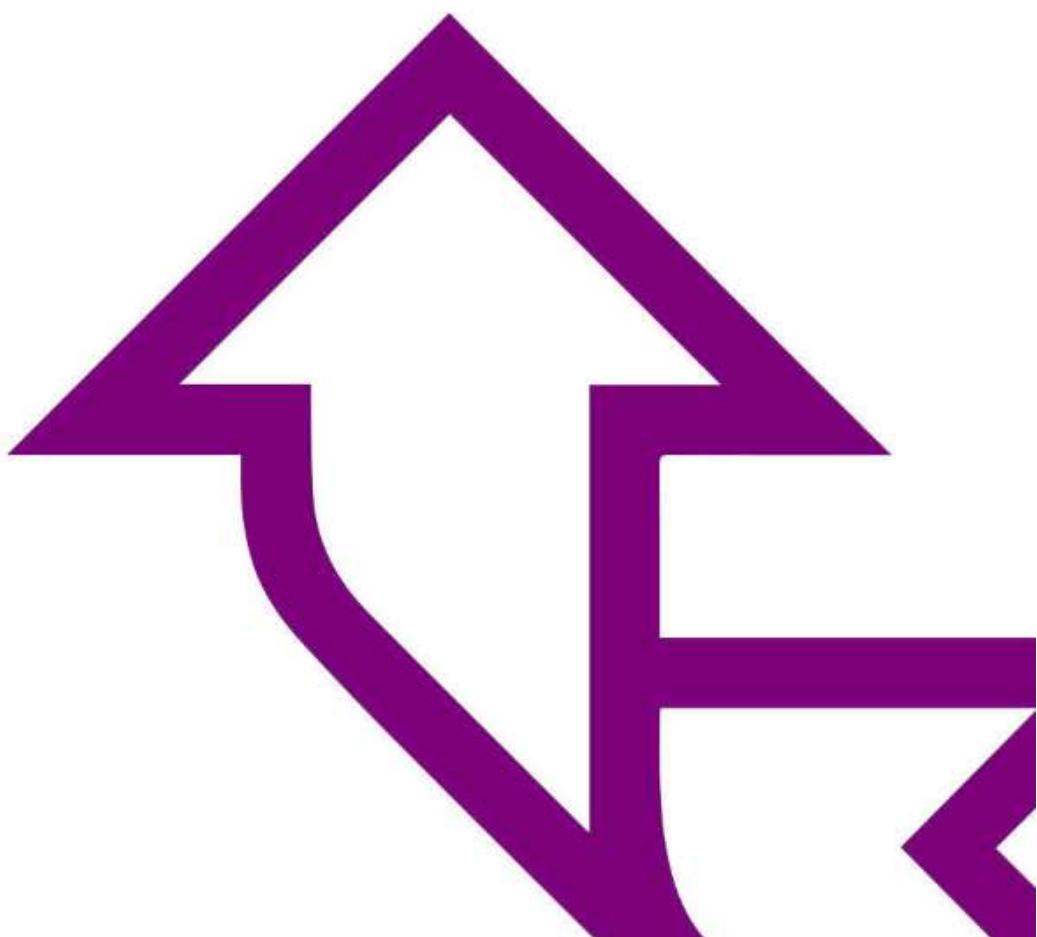
Simple maths and logical operators

Strings





Control Flow



OUTLINE

What is control flow?

Conditionals

- if/else if/else
- switch

Looping

- for
- while
- do ... while

Nesting statements

Debugging in IntelliJ



QA

OBJECTIVES

By the end of this session we should be able to:

- Write conditional and looping statements in Java
- Be able to step through out code with the debugger

What is control flow?

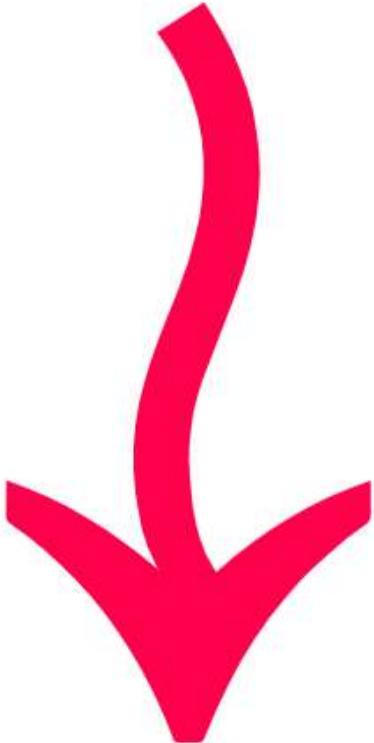
A program made up of single statements which print to the command line isn't much use in the real world

Control flow describes the ways the program runs

- Loops (do something five times)
- Conditionals (if something is true do this, otherwise do something else)

Uses the comparison and logical tests from the previous chapter

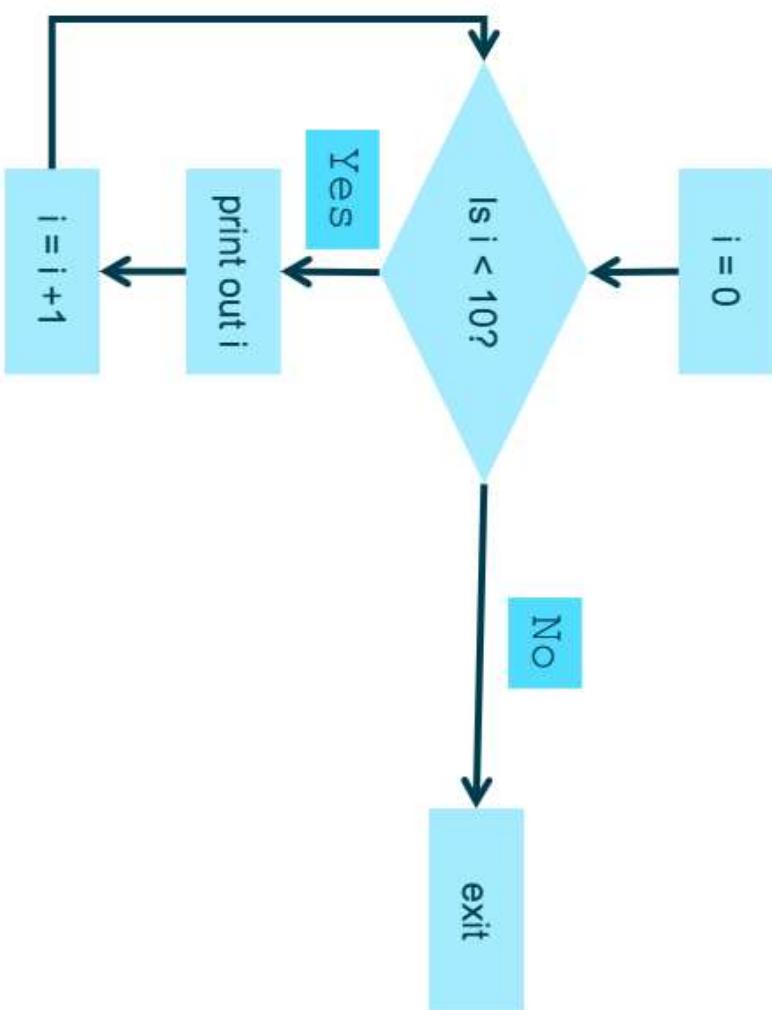
- Needs to evaluate an expression to be either true or false



Q1 Flow diagrams

Programs are a series of instructions

- We can draw these as flow diagrams to help design the code steps



Q4 Conditionals - if

If some condition is true, the code between the braces { ... } is run

```
if (condition) {  
    //... Do something ...  
}  
  
if (i % 2 == 0) {  
    System.out.println (i + " is Even");  
}  
  
if (i % 2 == 0 && i > 10) {  
    System.out.println (i + " is Even");  
    System.out.println (i + " is greater than 10");  
}
```

Q& A **if ... else**

If some condition is true, do something, otherwise do something else

```
if (condition) {  
    //... Do something ...  
} else {  
    //... Do something else ...  
}
```

```
if (i % 2 == 0) {  
    System.out.println (i + " is Even");  
} else {  
    System.out.println (i + " is Odd");  
}
```

Q4 if ... else if ... else

If we want to check for more than one state at a time

- We can nest if statements, but using 'else if' is safer

```
if (condition) {  
    //... Do something ...  
} else if (condition2) {  
    //... Do something else ...  
} else {  
    //... An all other cases do this  
}
```

```
if (i == 1) {  
    System.out.println ("i is 1");  
} else if (i == 2) {  
    System.out.println ("i is 2");  
} else {  
    System.out.println ("i is not 1 or 2");  
}
```

QA Ternary operators

A shorthand version of the if statement

```
(condition) ? expression1 : expression2;
```

If the condition evaluates to true then expression 1 is executed, otherwise expression 2 is executed

```
string str = (i % 2 == 0) ? "even" : "odd";
```

The two expressions must use the same type

- In this usage both 'even' and 'odd' are strings, so the statement will execute correctly

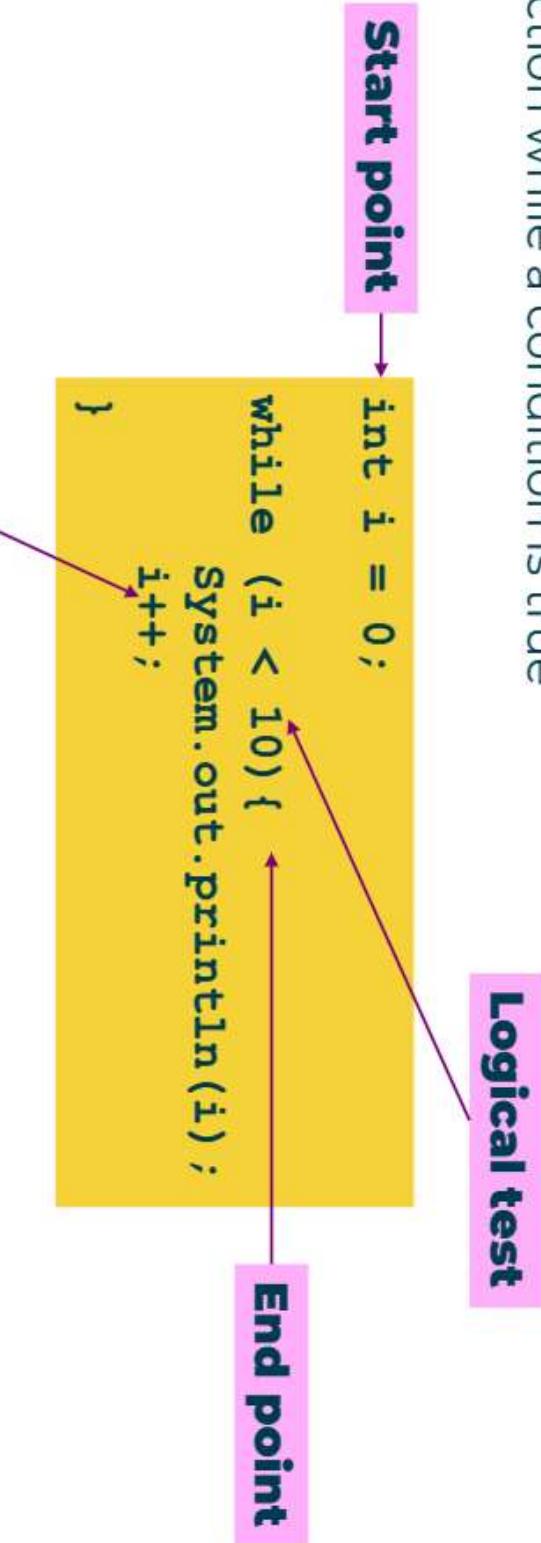
Qn switch

- Switch statements can have multiple possible execution paths
- Works with only certain data types byte, short, char, int, enumerated types, String
- **break** – stops execution from continuing onto the next case
- **default** – if none of the listed cases applies

```
int i = 5;
switch (i) {
    case 1:
        System.out.println("i is one");
        break;
    case 2:
        System.out.println("i is two");
        break;
    default:
        System.out.println("i is not one or two");
}
```

Q1 while

Do some action while a condition is true



Performs the test before executing any of the body of the loop

Does not need to be an integer!

Qn do... while

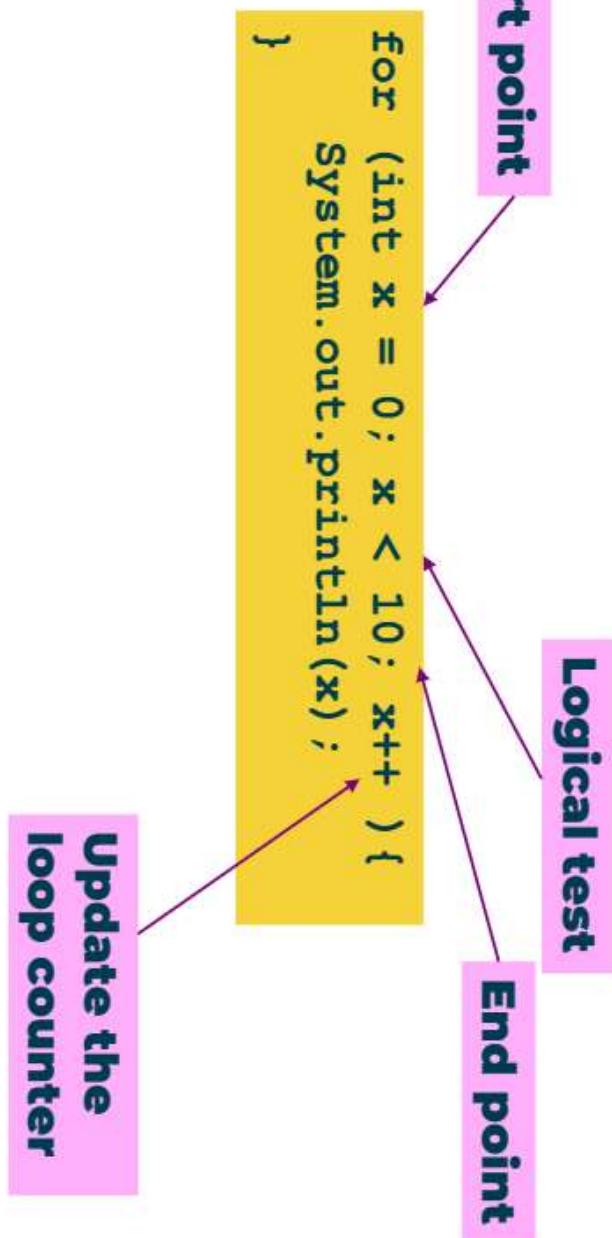
Similar to the while loop, but the check is performed at the end

- We can use this to ensure that the body of the code is always run at least once
- Not used very often

```
do {  
    System.out.println(i);  
    i++;  
} while (i < 0);
```

Q4 for loops

For loops contain the loop counter inside the expression



Q4 More than one loop counter

You can declare more than one loop counter in a for loop

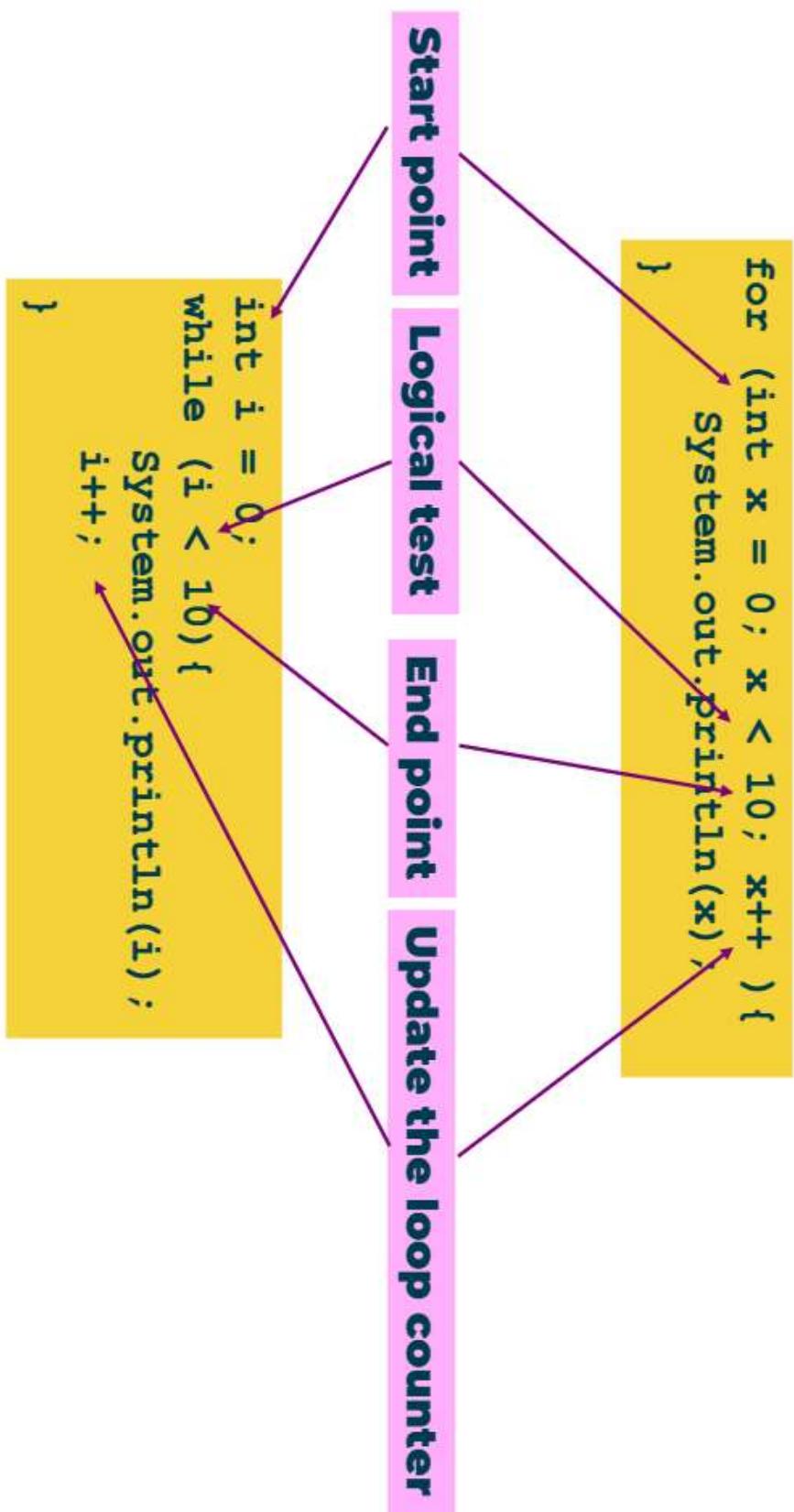
- The test is the same as required in a while loop with more than one counter used
- Remember to check both values if required

```
for (int i = 0, j = 10; i < 5 && j < 15; i++, j++) {  
    System.out.println("(" + i + ", " + j + ")");  
}
```

Output:

```
(0, 10)  
(1, 11)  
(2, 12)  
(3, 13)  
(4, 14)
```

Q4 Comparisons



Q1 Which to use when

while statements don't need to have the counter updated in the declaration

- They're better when you don't know how many iterations you will need
- Could never end – we need to include some test that will cause the loop to end
- Doesn't need to use numbers – we could test
 - Boolean value which is set in the loop
 - String value equal to something

for statements do have the counter

- Need to know the number of iterations ahead of time
- Can change the loop counter in the body – but it is considered very bad practice
- for loops always work with numbers!

Q4 Break out!

Sometimes you may not want to finish loop execution

The break keyword allows you to jump out of the loop at that point
- no matter if the condition was still true

```
int i = 0;  
while (true) {  
    if (i > 10) {  
        break;  
    }  
    i++;
```

This is not considered best practice as it can make the code harder to follow

Qn Looping through an array

- Often in your code you will want to loop through a data structure such as an array
- The array type has an attribute called 'length' which helps

```
int[] arr2 = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
for (int x = 0; x < arr2.length; x++) {
    System.out.println(arr2[x] + 100);
}
```

Using the length attribute ensures we use index values in the correct range for this array

Q4 Nesting statements

Nesting refers to putting one conditional / loop inside another one.

```
int i = 0;  
int j = 0;  
  
while (i < 10){  
    while (j < 10){  
        System.out.println("(" + i + ", " + j + ")");  
        j++;  
    }  
    j = 0;  
    i++;  
}
```

Nesting statements is necessary

- When an inner loop counter needs to cycle through its values for every value of an outer loop counter

QA Variable scope

Scope is the visibility of variables to the rest of the program

- In general, variables will only be visible within the braces where they were declared

```
public static void main(String[] args) {  
    ...  
    int i = 0; ↑  
    ...  
}
```

i is in scope

```
}
```

```
for (int j = 0; j < 10; j++) {  
    System.out.println(j); ↑  
    ...  
}
```

j is in scope

```
}
```

```
System.out.println(j); ↑
```

j is not in scope
This won't compile!



DEBUGGING IN INTELLIJ

QA Debugging in IntelliJ

As programs get more complicated it is harder to see where problems are

- The compiler can only tell us where we have syntax errors
- Logical errors are harder to spot!

IntelliJ (and other IDEs) have a built-in debugger which allows us to step through the program line by line and investigate what is happening

The code pauses at breakpoints

- Ways to create a breakpoint:
 - Click into the gutter to the left of the line of code
 - Press Ctrl+F8
 - Run menu → Toggle Breakpoint
 - Line Breakpoint



```
while(b < 10) {  
    while(c < 10) {  
        System.out.println("(" + b + ", " + c + ")");  
        c++;  
    }  
    c = 0;  
    b++;  
}
```

QA Debugging in IntelliJ

Use the Debug option, rather than Run

- Run → Debug 'ClassName'
- Shift+F9
- Click the debug icon



IntelliJ then runs to the first breakpoint

Tick shows that
this breakpoint
has been reached

```
45
46  ↪ while(b < 10) { b = 0
47   while(c < 10) {
48     System.out.println("(" + b + ", " + c + ")");
49     c++;
50   }
51   c = 0;
52   b++;
53 }
```

QA The Debug tool window

The screenshot shows the IntelliJ IDEA interface with the Debug tool window open. The code being debugged is `C_loops.java`, specifically the line `while(b < 10) { b = 0;`. The current step number is 1.

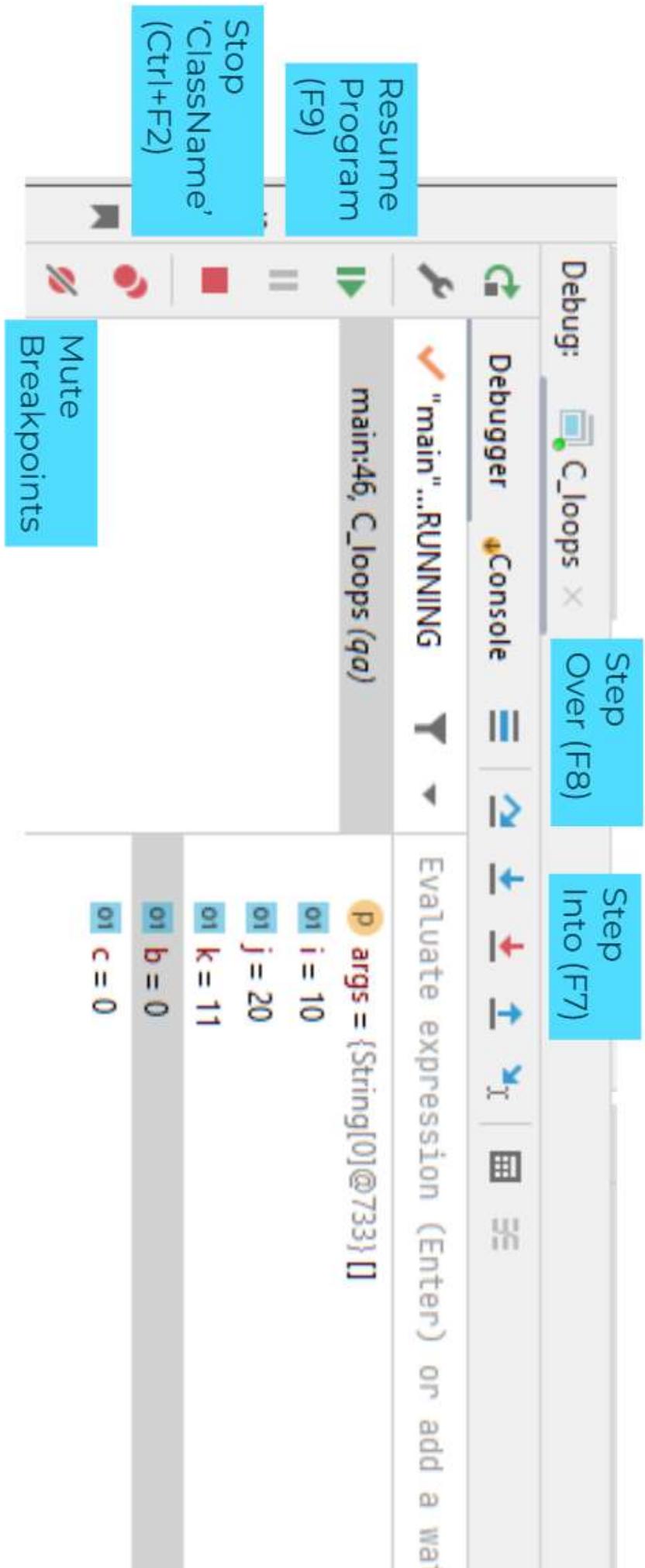
Console output tab: Shows the command `java C_loops` and the output of the program's execution.

Options to step through the program: Includes buttons for Step Into, Step Over, Step Out, and Evaluate Expression.

Current variables in memory: Displays the values of variables `i`, `j`, `k`, `b`, and `c`.

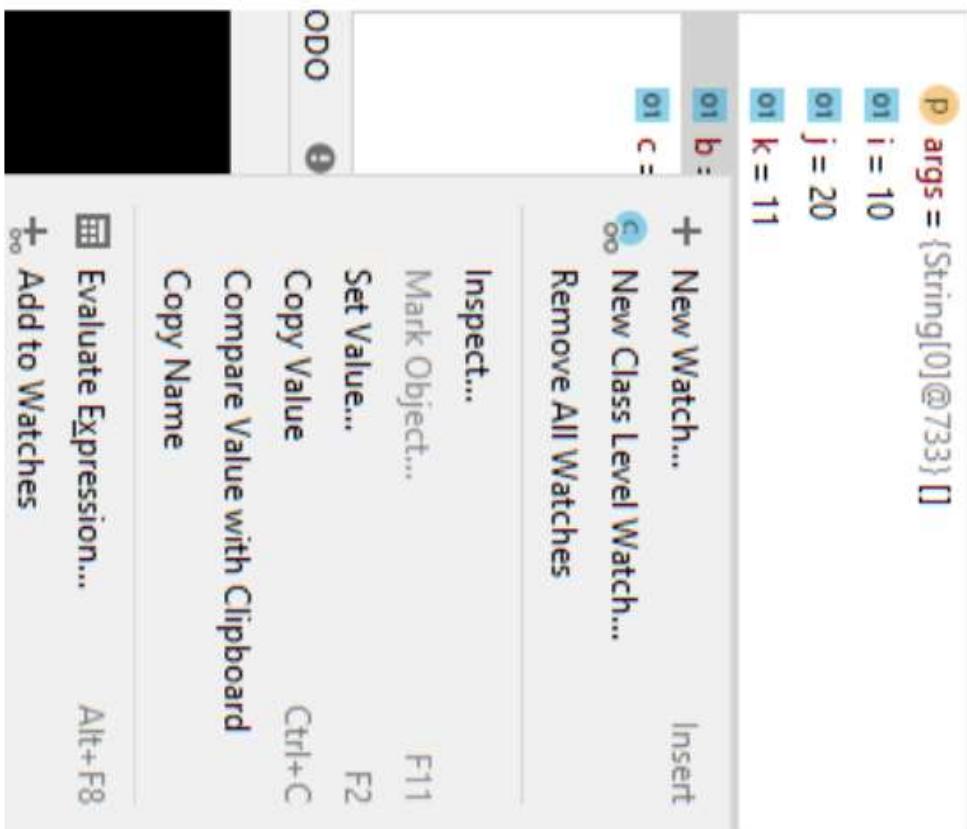
```
args = {"String[0]@733"}  
i = 10  
j = 20  
k = 11  
b = 0  
c = 0
```

QA Debugging – stepping through the code

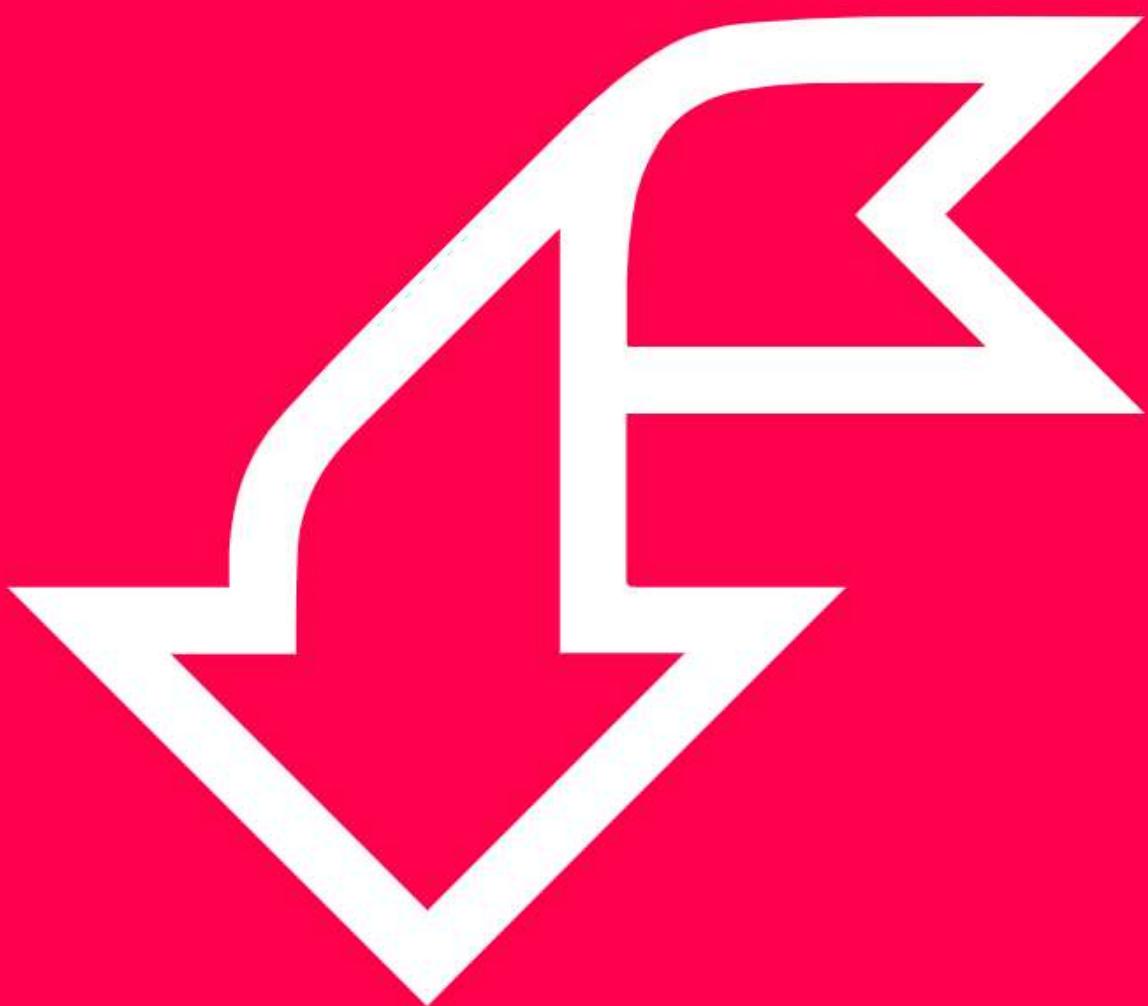


QA Debugging – watching variables

- In a program with a lot of variables it can be difficult to find the specific one we want to focus on
- We can choose individual variables and specify that we are interested in these no matter where we are in the program
- Right-click the variable we are interested in and select Add to Watches – it will be listed before other variables with a spectacles icon next to it
- We can also change variable values using Set Value (F2)



QA



Programming time

Practice with loops, conditionals, and
Boolean expressions

SUMMARY

What is control flow?

Conditionals

- if/else if/else
- switch

Looping

- for
- while
- do ... while

Nesting statements

Debugging in IntelliJ





Introduction to Objects



Java programming using IntelliJ IDEA

OUTLINE

What is an object?

Introduction to Java objects

- Defining a class
- Fields
- Methods

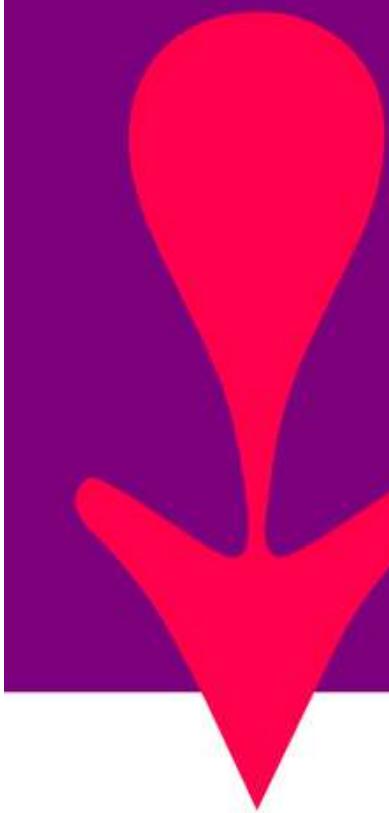
Creating objects

- Accessing fields and methods

Encapsulation

- Scope for variables and methods

Generating boilerplate in IntelliJ



OBJECTIVES

By the end of this session we should be able to:

- Describe a problem in terms of objects
- Write a class with fields and methods
- Instantiate objects of that class and call the methods
- Be able to describe what encapsulation is



Q4 Object-oriented programming

Models behaviour of a system using objects

- which group data and methods together in a logical format

Three main concepts:

- Encapsulation
 - Keeping related data and methods together
- Inheritance
 - Hierarchical structure of objects, being able to inherit methods from parents
- Polymorphism
 - Sub-classing allows for different behaviour of methods overriding the parent

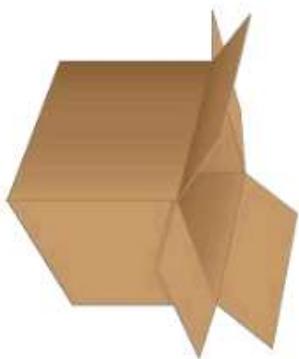
Q1 What is an object?

Objects in the real world:

- Car
- Tree
- Computer
- Desk
- Chair
- Rabbit

An object is 'something' that exists in the world

- Contains state and behaviour
 - Fields – information that describes the object
 - Methods – behaviour that the object can take



Qn Object example

For a Rabbit object

Fields – describe a Rabbit

- Name
- Colour
- Breed
- Age
- Current location

Methods – state what a Rabbit can do

- Hop
- Eat
- Mischief
- Increase age

QA

WHAT IS AN OBJECT?

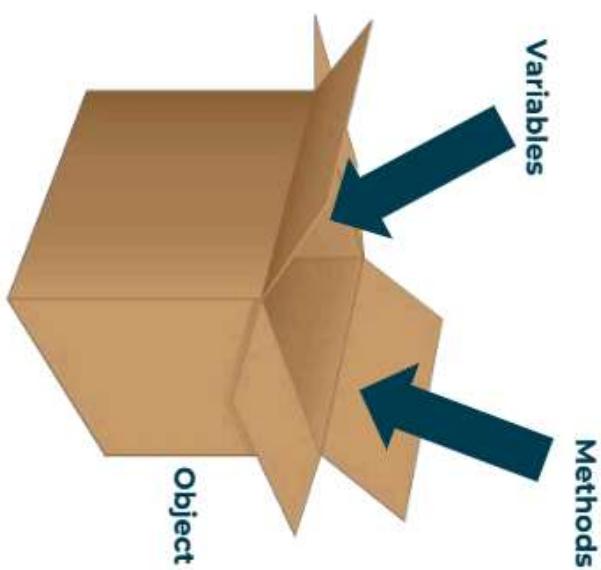
Q1 What are Java objects?

Java objects are instances of something

- Anything that could be a real world object can be a java object
- Contain fields which describe the object (variables)
- Contain methods which are actions the object can take

We can think of an object as being a box or container

- Fields and methods can be placed inside the box
- Methods can interact with anything else inside the box



QA A Java class

A Java class is a definition of an object

- It contains descriptions of the fields and methods
 - The class name must match the filename
- Naming convention is to capitalise the words in a name

```
public class Rabbit {  
    String name;  
    int age;  
    String colour;  
    public Rabbit(String n) {  
        name = n;  
    }  
    public void increaseAge () {  
        age = age + 1;  
    }  
    public int getAge () {return age; }  
}
```

The diagram illustrates the structure of a Java class named 'Rabbit'. A yellow rectangular box represents the class body. Inside, there are three main sections: 'Fields' (highlighted in pink), 'Constructor' (highlighted in pink), and 'Methods' (highlighted in pink). The 'Fields' section contains declarations for 'name' (String type) and 'age' (int type). The 'Constructor' section shows a constructor taking a 'String' parameter 'n' and setting the instance variable 'name' to it. The 'Methods' section contains two methods: 'increaseAge()' which increments the 'age' by 1, and 'getAge()' which returns the current value of 'age'. Red arrows point from each pink-highlighted label to its corresponding code block within the yellow box.

Q4 Fields

Declare the variables as you did before

Fields belong to the class they are declared within

- They are created when you create an instance of the class
- Their visibility can be changed to be public, private or protected
(We'll look at this later)

```
String name;  
int age;  
Object o;
```

Q1 The Constructor method

This method is called when the Java class is instantiated

- It has the same method name as the class name
- There is no return type (not even void)
- It tells Java how to initialise the object
 - Parameters passed in are used to set the values of fields

```
public class Rabbit {  
    String name;  
    int age;  
    String colour;  
    public Rabbit() {} // default construct  
    public Rabbit(String n, int age, String colour) {  
        this.name = n;  
        this.age = age;  
        this.colour = colour;  
    }  
}
```

Qn Other methods

Method declarations have five parts

- Accessibility
 - public
 - private
 - protected
- Return type
 - What type of variable the method will return
- Method name
- Parameters
- Method body
 - between the { ... }

```
public class Rabbit {  
    public int age;  
  
    public void increaseAge() {  
        age = age + 1;  
    }  
  
    public int getAge() {  
        return age;  
    }  
}
```

Q4 Method Parameters

Methods can have parameters passed into them

- They can then use those parameters as variables in the method
- They are not accessible outside this method
 - unless the value is saved to an instance variable

```
public double calculateCost(int numberofItems) {  
    double cost = (numberofItems * costPerItem);  
    double addVAT = (cost * 0.20) + cost;  
    return addVAT;  
}
```

We need to give the type of each parameter and a name

Multiple parameters are separated by commas

Q A The `toString()` method

Whenever you print out an object it's `toString()` method is implicitly called

If there isn't one, then a memory reference for the object is given

Output:

Rabbit@2760e8a2

To override this behaviour create your own `toString()` method in your class and have it return a String value to represent the contents of this object

```
public String toString() {  
    return "Name: " + name + " Age: " + age;  
}
```

Output:

Name: Peter Age: 2

CREATING OBJECTS

Q1 Instantiating an object

The class file is just the description of how the object is to be created and what it contains

We need to instantiate the class and create an object before we can use it

- This uses the keyword **new**
- Calls the constructor for the class

```
Rabbit roger = new Rabbit();
```

There is only one class definition, but there can be many objects created from that class

Q4 Instantiating an object

Rabbit class definition

```
public class Rabbit  
{  
    String name;  
    ..  
}
```

Instantiating Rabbit objects

```
Rabbit flopsy = new Rabbit();  
Rabbit mopsy = new Rabbit();  
Rabbit cottontail = new Rabbit();  
Rabbit peter = new Rabbit();
```

Q4 Using methods

To call methods on an object we use the dot (.) notation

- We've seen this before with array.length and System.out.println(...)

objectName.methodName(parameters)

So in the case of the rabbit it would be:

```
Rabbit flopsy = new Rabbit();
String str = flopsy.toString();
System.out.println(flopsy.getAge());
```

QA

ENCAPSULATION

Q4 Encapsulation

Keeping an object's data hidden and providing methods to allow appropriate access

There are four levels of protection we can give methods and fields

- public
- protected
- package wide (no modifier)
- private

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

Q4 Encapsulation

Generally all **fields** should be declared as **private**

- Otherwise the state of the object could be changed by code outside the class
- Allows for data validation on input and formatting on output

We then use **accessor and mutator methods** to read and change the data

- Also known as getter and setter methods

```
private String name;  
public void setName(String name) {  
    this.name = name;  
}  
  
public String getName() {  
    return name;  
}
```

Qn Static

We've seen that we need to instantiate a class to create an object before we can call methods on it

So how does the main method run?

```
public static void main(String[] args) {  
    ...  
}
```

The keyword here is **static**

- Static methods and variables can be called without an instance of the object being created
- They should not hold state
- They are not associated with an object

QA

GENERATING BOILERPLATE IN INTELLIJ

Qn Using IntelliJ to generate boilerplate

Boilerplate code refers to simple methods, classes and annotation that you will end up writing repeatedly

IntelliJ can automatically generate these for you

- Right click the Class Definition -> **Generate** (Alt+Insert)
- From the Generate context menu

Constructor

- lets you choose the fields to be initialised

Getter and Setter

- lets you select fields to create methods for

toString()

- will create a `toString()` method with the fields you choose

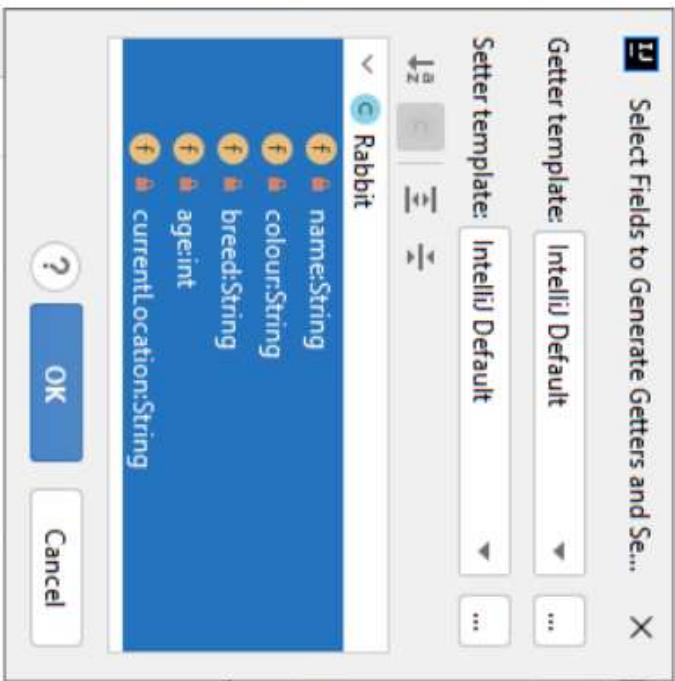


Q4 Generating Boilerplate



```
public Rabbit(String name,  
String colour,  
String breed,  
int age,  
String currentLocation) {  
    this.name = name;  
    this.colour = colour;  
    this.breed = breed;  
    this.age = age;  
    this.currentLocation = currentLocation;  
}
```

Q4 Generating Boilerplate



```
public String getName() {  
    return name;  
}  
public void setName(String name) {  
    this.name = name;  
}  
public String getColour() {  
    return colour;  
}  
public void setColour(String colour) {  
    this.colour = colour;  
}  
...  
public void setCurrentLocation(String currentLocation)  
{  
    this.currentLocation = currentLocation;  
}
```

Q4 Generating Boilerplate



```
public String toString() {  
    return "Rabbit{" +  
        "name=" + name + '\'' +  
        ", colour=" + colour + '\'' +  
        ", breed=" + breed + '\'' +  
        ", age=" + age +  
        ", currentLocation=" +  
        currentLocation + '\'' +  
    } ;  
}
```

Exercise

Create classes and some objects

- Remember to keep fields private
- Getter and Setter methods are used to access fields
- Methods should be public if you want them to be visible outside the scope of the class
- IntelliJ tools can auto generate methods, but try writing them yourself first!

SUMMARY

What is an object?

Introduction to Java objects

- Defining a class
- Fields
- Methods

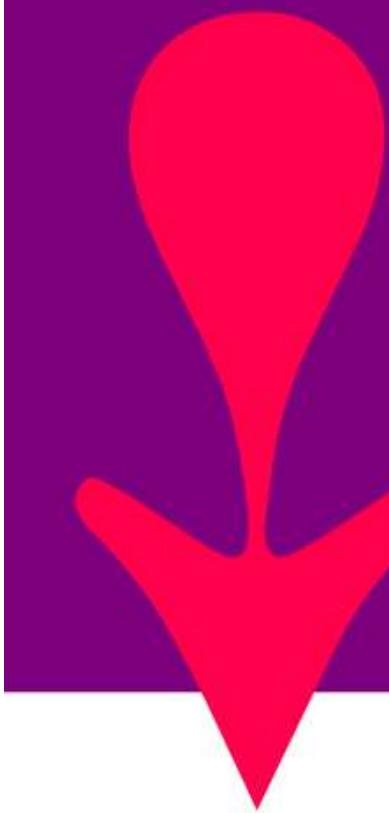
Creating objects

- Accessing fields and methods

Encapsulation

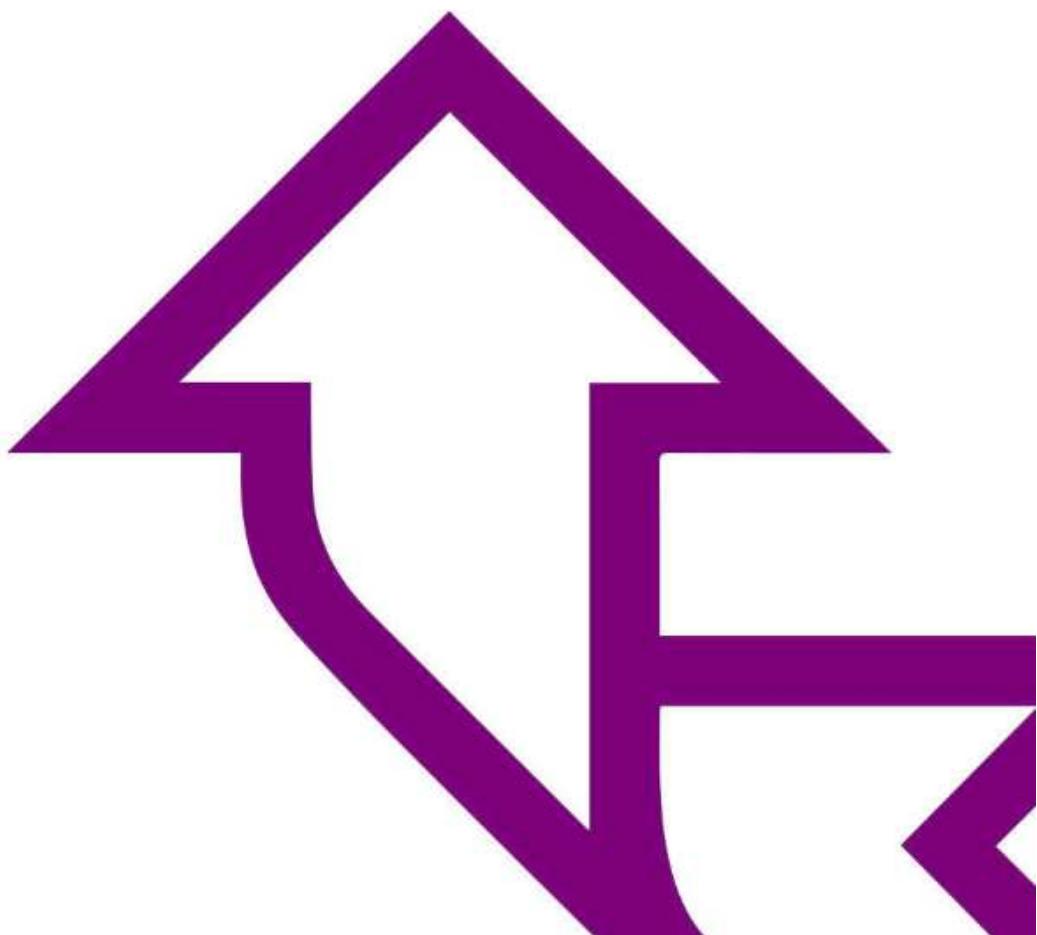
- Scope for variables and methods

Generating boilerplate in IntelliJ





Inheritance and Polymorphism



OUTLINE

Inheritance

- What is inheritance?
- Inheritance in Java

Object hierarchies

- abstract
- extends



QA

OBJECTIVES

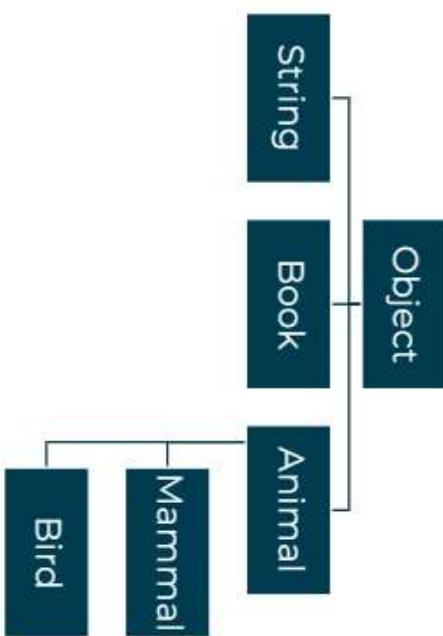
By the end of this session we should be able to:

- Describe what inheritance is in object oriented programming
- Be able to use inheritance and abstract classes in Java

Q1 What is inheritance?

Inheritance in the real world is the act of receiving something or other

- Property, titles, money after people pass away
 - Biological inheritance – genes passed down through families
- Inheritance is one of the key concepts in object-oriented programming
- Objects can inherit fields and methods from other objects
 - This puts everything in an object hierarchy



Q1 Why use inheritance?

Code reuse

- If every class has a 'calculate price' method then this should be written once rather than in five places

- Easier to debug code

- Easier to update code

Express relationships between objects

- 'is a' relationship

- A cat is a mammal which is an animal, which is an object (in our program)

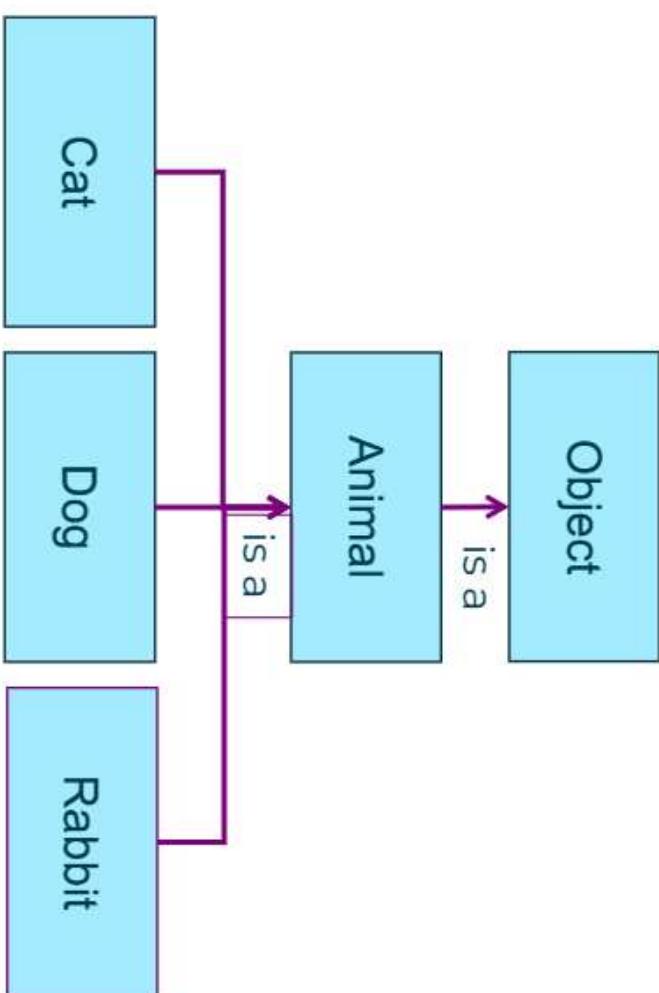
Using inheritance, classes can be structured in a hierarchy

- Child classes can access fields and methods in the parent object
- If they are public or protected visibility

Parent classes do not know about their children, so can't access any of their methods

- More specific implementations of classes are further down the tree

Class hierarchy



Q4 Extending a class

To create a sub-class we use the keyword **extends**

The sub-class implementation can have additional fields and methods to the parent class

```
public class Animal {  
    private String name;  
    private int age;  
  
    public Animal  
        (String name, int age) {  
            this.name = name;  
            this.age = age;  
        }  
    }  
  
public class Cat extends Animal {  
    boolean fluffy;  
  
    public Cat(String name,  
              int age,  
              boolean fluffy)  
    {  
        super(name, age);  
        this.fluffy = fluffy;  
    }  
    public boolean isFluffy() {  
        return fluffy;  
    }  
}
```

Qn Inheritance in Java

- Uses the **extends** keyword

- Only supports **single inheritance**

- To access methods in the parent class use the **super** keyword

Every object in Java inherits from the **Object** class

- Provides basic implementations of methods for all classes to inherit

toString()

hashCode()

equals(Object o)

- The Object class implementation of the **toString()** method returns

ClassName@hashRepresentationOfTheClass

Rabbit@4c873330

We override this functionality when declaring our own **toString()** methods

Qn Access parent class methods and fields

Use the keyword **super** to access a parent object's methods and fields

In the superclass:

```
@Override  
public String toString(){  
    return "Name: " + name + " Age: " + age;  
}
```

In the subclass

```
@Override  
public String toString(){  
    return "Cat: " + super.toString();  
}
```

Qn Public / protected / private visibility

Public

- The entire world has access including sub-classes

Private

- No one outside the class can access the method or field including subclasses

Protected

- Sub-classes can access this method or field
- Convention is to use it on methods, and use the accessor methods to access the field values
- Classes in the same package can access this method or field (even if not sub-classes)

Qn Abstract classes and methods

Abstract classes allow for the parent in a relationship to describe what methods should be there without providing an implementation

- **abstract methods** – have no body
- This class can't be instantiated into an object
- Uses the keyword **abstract** in the class definition

Sub-classes must @Override the behaviour of the abstract methods from the parent class

```
public abstract class Shape {  
    public abstract double getArea();  
}
```

Qn Abstract classes and methods

Subclasses must provide the method body

- If this is not done then the subclass will not compile
- The @Override annotation tells us the JVM will use this implementation rather than the parent's method – gives a warning if the child method class signature doesn't match the parent's

```
public class Rectangle extends Shape{  
    @Override  
    public double getArea () {  
        return height * width;  
    }  
  
}  
  
public class Circle extends Shape {  
    @Override  
    public double getArea () {  
        return Math.PI * Math.pow(radius, 2);  
    }  
}
```

Qn overriding behaviours

We can override the behaviour of any method in a superclass this way, not just abstract methods

- We've seen this with the `toString()` method that we are overriding from the `Object` class

In superclass:

@Override

```
public String toString() {  
    return "Name: " + name + " Age: " + age;  
}
```

In subclass:

@Override

```
public String toString() {  
    return "Cat: " + super.toString();  
}
```

Qn Overloading methods

What if we want to change the behaviour of a method based on what we pass it?

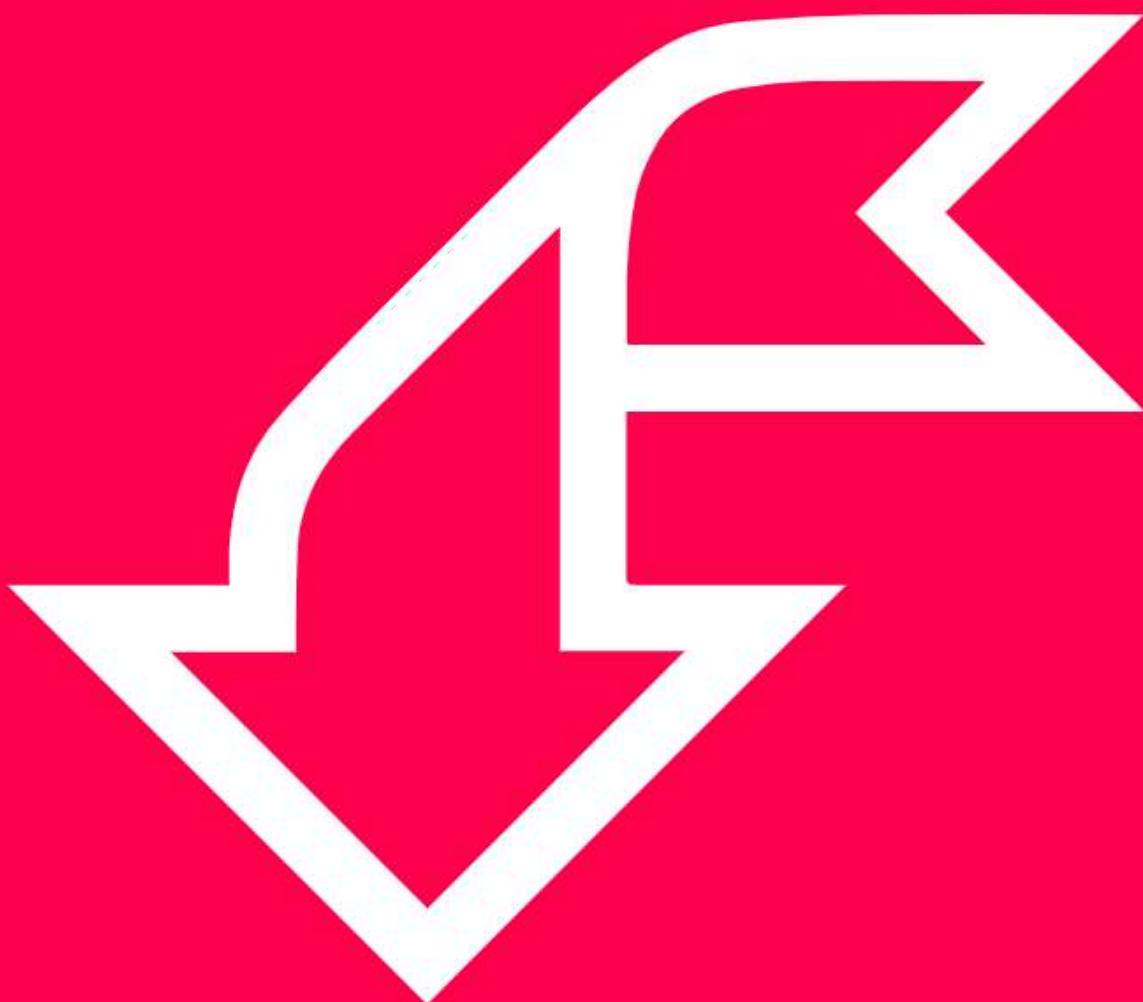
- Overloading allows us to have different implementations of the same named method
 - With different parameters

```
public String sayHello() {  
    return "Hello!";  
}  
  
public String sayHello(String name) {  
    return "Hello " + name;  
}
```

Qn Overloading constructors

Overloading is also useful for constructors

```
public Book (String ID,  
           String name,  
           String[] authors,  
           double price)  
{ ... }  
  
public Book ()  
{ /* provide defaults for everything */ }  
  
public Book (String ID,  
           String name)  
{ /* provide defaults for some fields */ }
```



Exercise

Create objects that inherit fields and methods from a parent

Use inheritance and abstract classes in Java

SUMMARY

Inheritance

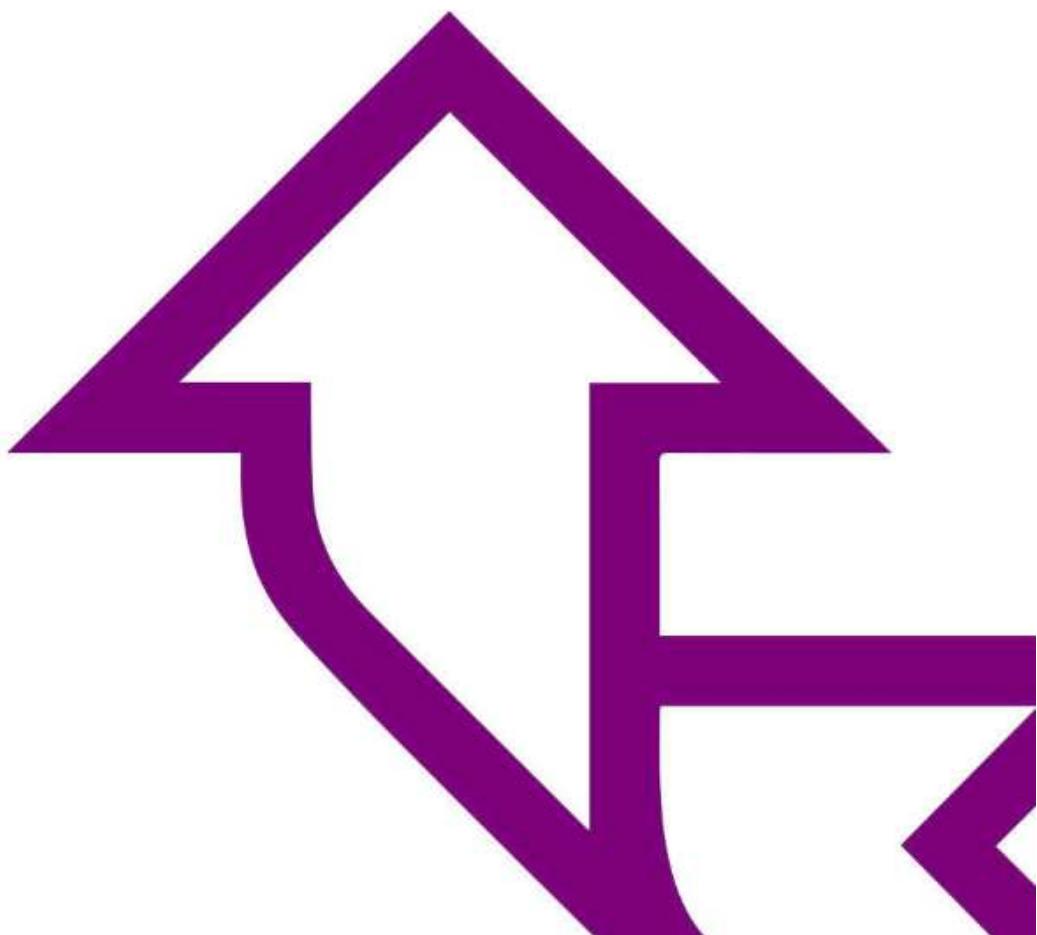
- What is inheritance?
- Inheritance in Java

Object hierarchies

- abstract
- extends



Java Interfaces



OUTLINE

Java interfaces

- Implementing interfaces
- Extending interfaces

QA

OBJECTIVES

By the end of this session we should be able to:

- use interfaces to give objects particular sets of methods
- create an interface by extending another interface

Q1 What is a Java interface?

A Java interface is a specification of method signatures

Interface represents a pure abstract class concept

Java only allows for single inheritance

Interfaces are a way of ensuring an object has a set of behaviours without needing to inherit from more than one class

- Interfaces define method names, return types and parameters required
- Classes that implement an interface supply the method body and code
- A class can implement more than one interface!

Fields in interfaces are implicitly public static and final

- Interfaces do not have their own state
- You can't create an object from an interface

Q4 Implementing interfaces

Declare an interface using the `interface` keyword, rather than class

```
public interface GuideDog {  
    public String crossRoad();  
    public boolean working();  
}
```

Classes can then implement this interface

```
public class Dog extends Animal implements GuideDog {  
  
    @Override  
    public String crossRoad() {  
        //implementation here  
    }  
  
    @Override  
    public boolean working() {  
        //implementation here  
    }  
}
```

Q4 Extending interfaces

Interface can extend one or more interfaces at a time

```
public interface GuideDog {  
    public String crossRoad();  
    public boolean working();  
}
```

```
public interface RetiredGuideDog extends GuideDog{  
    public String retirement();  
    public boolean isRetired();  
}
```

Class which implements RetiredGuideDog should override the methods of GuideDog and RetiredGuideDog otherwise the class itself should be declared as abstract

Q4 Implementing a child interface

```
public class Dog extends Animal implements RetiredGuideDog {  
    @Override  
    public String crossRoad() {  
        //implementation here  
    }  
    @Override  
    public boolean working() {  
        //implementation here  
    }  
    @Override  
    public String retirement() {  
        //implementation here  
    }  
    @Override  
    public boolean isRetired() {  
        //implementation here  
    }  
}
```

Qn Extends vs. implements

Extends is object inheritance

- Object exists in a hierarchy
- Can only extend from one class
- Contain abstract method, concrete methods and fields
- Explains what an object is

Implements uses interfaces

- Can implement many interfaces
 - Theoretical maximum of 65535
- Contain unimplemented methods only
- No internal state
- Describe what an object should do

Q& What class am I?

Each object knows what class it is when it is created

- We can access this using the **getClass()** method

```
Dog d = new Dog ("Spot", 2);  
  
System.out.println(d.getClass()); // class com.other.Dog
```

We can also see if an object is an **instanceof** another class

- This also works for checking if the class implements a particular interface

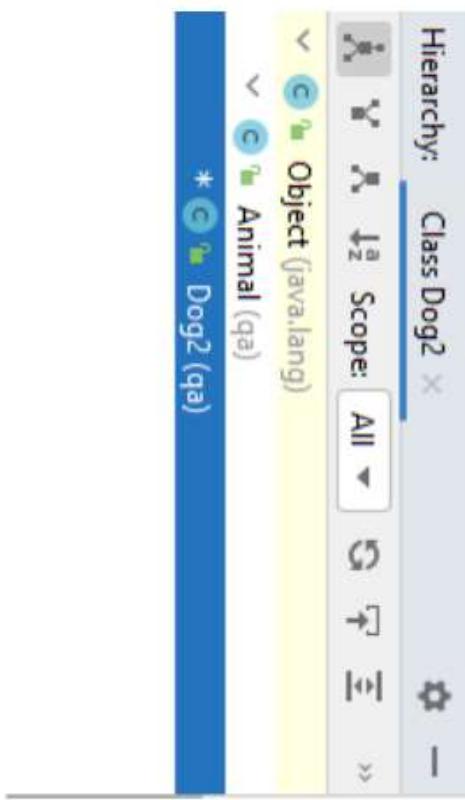
```
if (d instanceof Dog) {  
    System.out.println("It's a dog"); // It's a dog  
  
}  
  
if (d instanceof GuideDog) {  
    // It's a Guide Dog!  
    System.out.println("It's a Guide Dog!");  
}
```

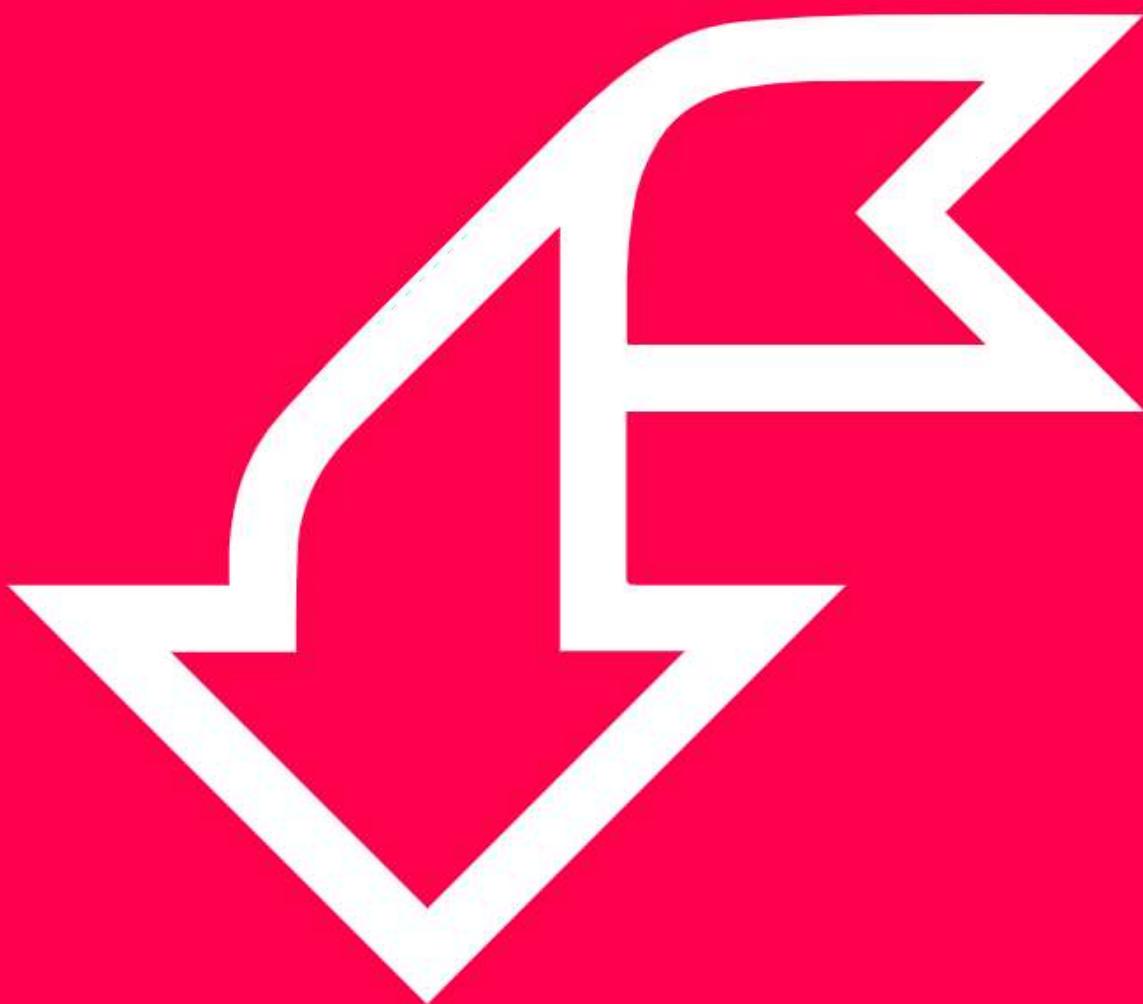
QA The type hierarchy in IntelliJ

IntelliJ can visualise how classes are connected to one another

Type Hierarchy:

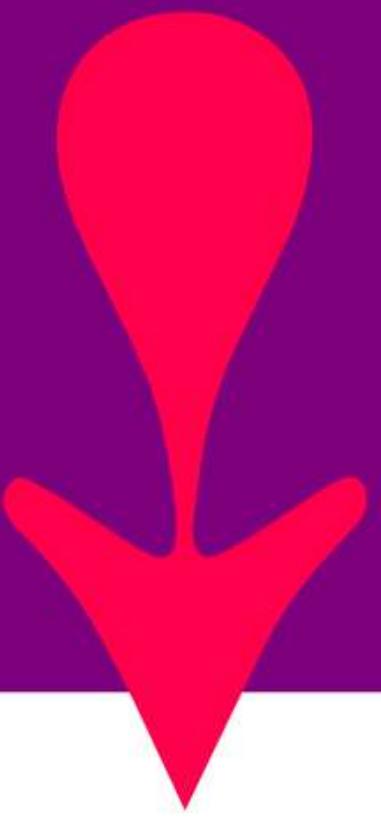
- Right-click on the class in the Project tool
- > Browse Type Hierarchy (Ctrl+H)





Exercise

Use interfaces to give the objects extra functionality

A stylized red flower logo with five petals, centered on a white background.

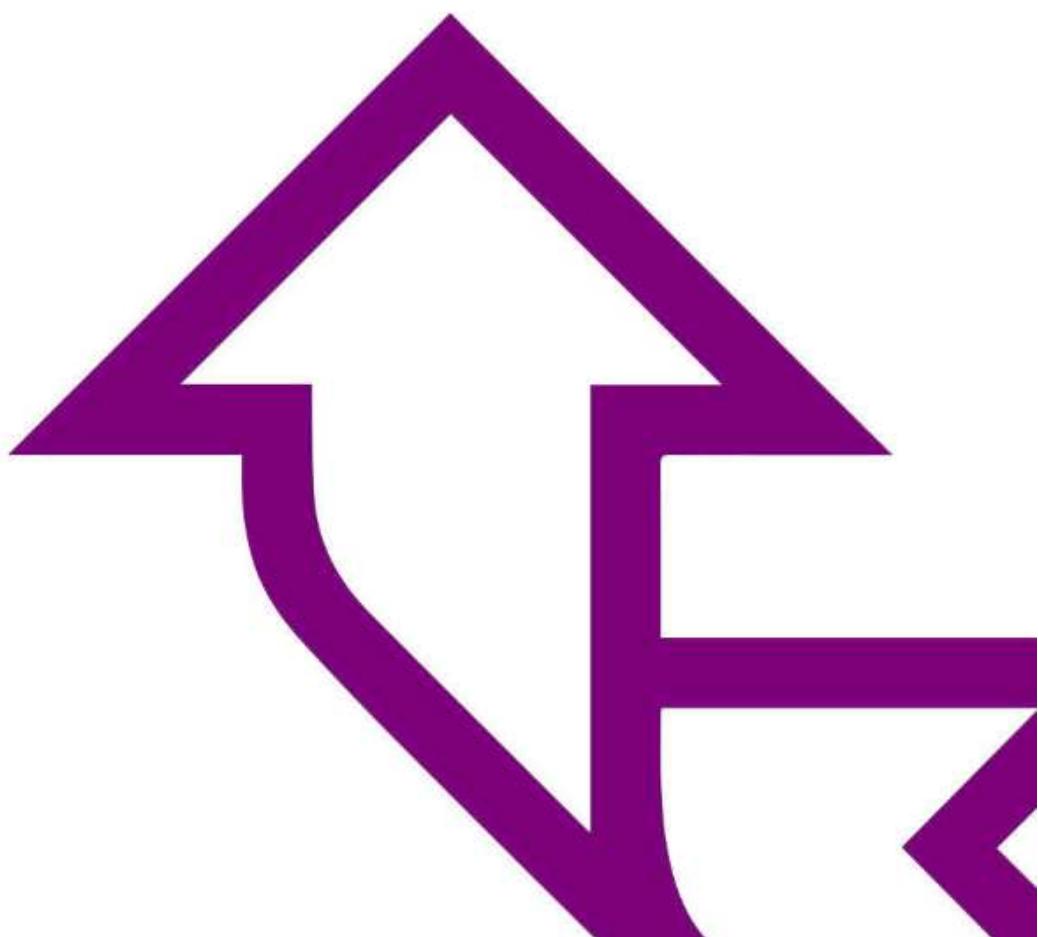
QA

SUMMARY

Java interfaces

- Implementing interfaces
- Extending interfaces

Collections



OUTLINE

Basic collection classes

- Lists
- Maps
- Sets

Using collections

- Iterating over collections
- Sorting
- Comparators

Generics

- Bounded types



OBJECTIVES

By the end of this session we should be able to:

- use a variety of different collection classes
- be able to sort a collection class without writing our own sort method
- understand and be able to use generics

Q1 What are the collection classes?

The collection classes group multiple objects together into a single unit

- List of references in memory to where objects are stored

Used to store similar things together

- A list of all the books in a shop

- Much like an array,

but there are more methods available for the Collection classes

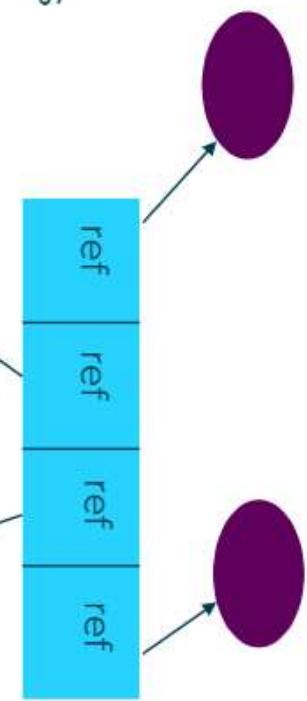
Different Interface types and concrete classes

- **Lists** – ArrayList, Vector, Stacks

- **Maps** – EnumMap, HashSet, HashMap, TreeMap

- **Sets** – EnumSet, HashSet, TreeSet

- **Queues** - DelayQueue, LinkedBlockingDeque, LinkedBlockingQueue



QA Basic collection methods

Method	Description
<code>int size()</code>	Returns the number of elements in the collection
<code>boolean isEmpty()</code>	Returns if the collection is empty
<code>boolean contains (Object o)</code>	Does the collection contain this object
<code>boolean add (Object o)</code>	Adds the object
<code>boolean remove (Object o)</code>	Remove the object that matches this one
<code>Iterator<E> iterator</code>	Returns an iterator we can use to go through all the elements in the collection
<code>boolean containsAll(Collection c)</code>	Does the collection contain all of these items
<code>boolean addAll(Collection c)</code>	Add all the items at once
<code>boolean removeAll(Collection c)</code>	Remove all the items at once
<code>boolean retainAll(Collection c)</code>	Remove everything except these items
<code>void clear()</code>	Empty the collection
<code>toArray()</code>	Converts the collection into an array

Q Importing the collection classes

To use any of the collections we need to import them into the project

- The import statement comes after the package statement, but before the class statement
- This adds the ability to create objects and call methods from classes in other packages

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

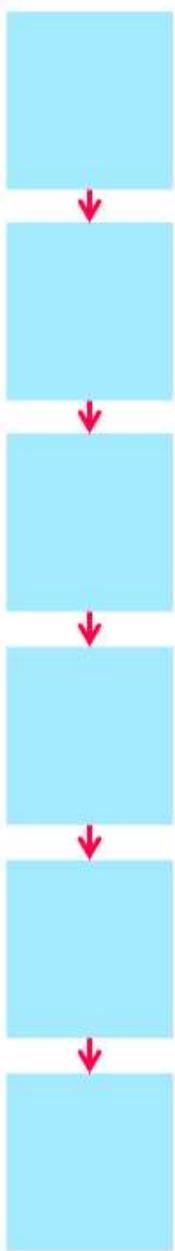
IntelliJ can sort out the imports for you

- Alt+Enter when 'Cannot resolve symbol ArrayList' occurs
- Puts the import statement in

Q& Lists and ArrayList

A list is an ordered collection

- Stores elements one after another in the order they were added



ArrayLists can be of any size

- It does not need to be specified up front
- The type of the ArrayList elements is specified in the angle brackets <>
- We do not need to specify the object type after the new statement

```
ArrayList<Object> arrayList = new ArrayList<Object>();  
// since java 7  
ArrayList<Object> arrayList = new ArrayList<>();
```

Qn Adding to an ArrayList

Use the .add(Object o) method to add a new element to the ArrayList

- This is added to the end of the list
- The list is automatically resized for you as required
- Either add the object directly or use the object name as a reference

```
Book b = new Book ("Amazing Book", new String[5], 10.00);  
ArrayList<Book> bookList = new ArrayList<>();  
bookList.add(new Book("Title", new String[5], 19.00));  
bookList.add(b);
```

There is also an option to add at a specific index

- The object is inserted at that position
- Objects are renumbered after the new one as required

```
bookList.add(index, element);
```

Qn Accessing an item from an ArrayList

We access the elements of the ArrayList using the **.get()** method

- The index of the element is required

bookList.get(index)

If we want to find a specific object in the list (a book with a specific title)

- Use the **.contains()** method to establish if it is in the collection
- Iterate through the collection and on finding the correct book returning that index

```
for (Book book : bookList) {  
    if (book.getName() .equals ("Amazing Book")) {  
        System.out.println (  
            "Index of Amazing Book: " +  
            bookList.indexOf(book));  
    }  
}
```

Q4 Removing Elements from an ArrayList

The **.remove()** method drops items out of the list

- Using the object as a reference
- Or the index

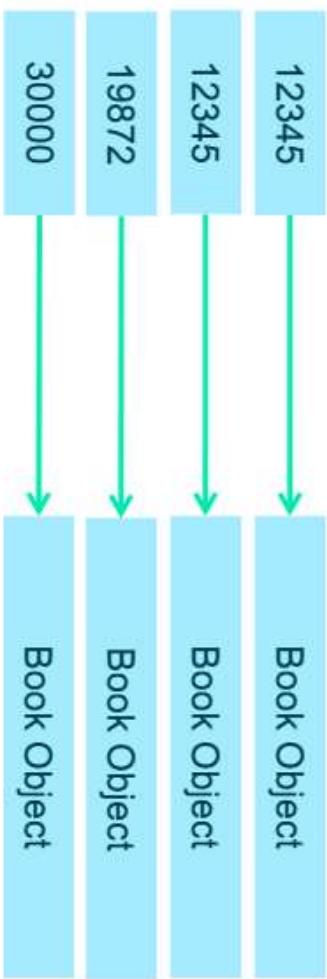
```
bookList.remove(0);  
bookList.remove(book);
```

Qn Arrays vs ArrayLists

Array	ArrayList
Fixed Size	Variable Size
Fixed Memory	More memory required
.length used to find size	.size() method
[index] to access elements	.get(index) to access elements
Arr[index] = 5 to update or add	.add() to add new elements
Updating elements overwrites the old one	Updating elements requires removing the old one
Need to recreate the array to remove elements	.remove() to remove old elements
	More helper methods available

Qn Maps

Maps link keys to values



Data is stored according to the key, so we can go to an object stored in a map directly

- In a list we need to loop through from start to end looking for an object

Keys can be of any type, but must be unique

Three general purpose map implementations provided in Java

- HashMap
- TreeMap
- LinkedHashMap

Qn **HashMap**

Implemented using a hash table

- There is no order to the keys or the values
- A hash value is computed from the key
- The value is stored in memory based on that hash
- To retrieve an item the key is hashed again, and the item retrieved directly from memory

To create a HashMap we need to tell it what type of object we are using for the Key and the Value

- Both the key and value must be objects, not primitives

```
HashMap<KeyType, ValueType> map = new HashMap<>();
```

```
HashMap<String, Book> bookMap = new HashMap<>();
```

Q4 Using HashMaps

We use the put method to add something to a hash map

```
map.put("abcde", new Book());
```

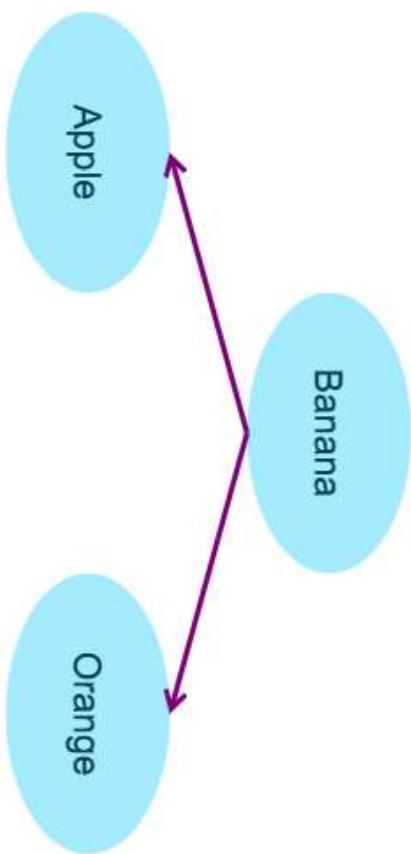
We can then get different pieces of information from the hash map

Name	Return type	Example
get	Object	map.get("abcde")
isEmpty	Boolean	map.isEmpty()
keySet	Set<KeyType>	map.keySet()
remove	Object	map.remove("abcde")
replace	Previous Object stored in that key	map.replace("abcde", new Book())
contains	Boolean	map.contains("abcde")
values	Collection<ValueType>	map.values()

QA TreeMaps and LinkedHashMaps

TreeMaps store information in a tree structure

- Order is based on the Key or a Comparator



LinkedHashMaps

- Creates a linked list of the elements in the map
- Allows you to get elements back in the order they were added

QA Sets

Sets contain no duplicate elements

- So it could not contain two strings 'abcde' or two references to the same object
- Based on the mathematical definition of a set

Behave in a similar way to other collections

Four types of set in Java

- HashSet
- LinkedHashSet
- TreeSet
- EnumSet

QA Iterators

All the collection classes implement the Iterable interface

- This gives the classes an iterator we can use to loop over all the elements in the collection
- Iterators have three methods:

- **hasNext()** – returns a boolean based on if we are at the end of the collection
- **next()** – get the next item
- **remove()** – remove the item from the underlying collection

Q4 Using iterators

Use the iterator() method to get an Iterator for the collection

- Give it the type you want to use, this will match the type of objects stored in the collection

```
Iterator<Book> iter = bookList.iterator();  
  
while (iter.hasNext ()) {  
    System.out.println(iter.next());  
}  
  
Iterator<Book> iter2 = bookMap.values().iterator();  
  
while (iter2.hasNext ()) {  
    System.out.println(iter2.next());  
}
```

Qn Processing collections elements

- You can also use the standard for and while loops to step through elements
- The **.size()** method returns the size of the collection
- Also a ‘for each’ style loop – the enhanced for loop – a more convenient syntax
 - No loop counter
 - Goes through each element in the collection in turn
 - Depends on the type of collection as to which order the elements are processed in
 - We will see the functional forEach() method later

```
ArrayList<Book> bookList = ... ;  
  
for (Book b : bookList) {  
    ... Can use b here ...  
}
```

QA Sorting

- Sorting collections is a useful and often essential feature
- Some collections sort themselves by using the key or comparator
- You never need to write your own sorting method
- Java has some built-in sorting methods for us
- Some objects have natural sorting
 - Alphabetically
 - Numerically
- For other objects we need to tell it how to compare them

```
ArrayList<Integer> intList = new ArrayList<Integer>();  
  
Collections.sort(intList);  
  
for (int x = 0; x < intList.size(); x++) {  
    System.out.print(intList.get(x) + ", ");  
}
```

Q4 Implementing the comparable interface

The **comparable interface** has one method: **compareTo()**

- This tells the object whether it is the same as, less than or greater than another object. The designer decides on the implementation of this method

```
public class Book implements Comparable<Book> {  
    ...  
    @Override  
    public int compareTo(Book that) {  
        if (this.price < that.getPrice()) {  
            return -1;  
        } else if (this.price > that.getPrice()) {  
            return 1;  
        } else {  
            //if this equals that  
        }  
    }  
}
```

Qn Comparators

You may not want to implement the Comparable interface in your class, or you may need alternative sort criteria

- We can create Comparator helper classes to help with the sorting
- Each has a compare() method

```
public class CompareBooks implements Comparator<Book> {  
    @Override  
    public int compare(Book book1, Book book2) {  
        if (book1.getTitle().compareTo(book2.getTitle()) < 0) {  
            return -1;  
        } else if (book1.getTitle().compareTo(book2.getTitle()) > 0) {  
            return 1;  
        } else {  
            return 0;  
        }  
    }  
}
```

Usage:
`Collections.sort(bookList2, new CompareBooks());`

QA Generics

Generics allow classes to be instantiated with different object types

- Allows for type to be a parameter
- Uses the angle bracket notation <>
- We've already used this with the Collections

```
HashMap<String, Book> map;  
ArrayList<BankAccount> list;
```

We can create Collections without giving a type

- Any type of object can be stored in the list
- When we get the objects from the list we can't guarantee that they will be the right type
- Generics allow us to define classes without worrying about the type

Q4 Generic object example

```
public class GenericObject<T> {  
    T genObject;  
  
    public GenericObject(T genObject) {  
        this.genObject = genObject;  
    }  
  
    public void setObject(T genobject) {  
        this.genobject = genobject;  
    }  
  
    public T getobject() {  
        return genObject;  
    }  
}
```

Q4 Generic methods

- These introduce their own generics at the method level
- You add the method type in triangular brackets after the scope declaration

```
public <T> void doSomething (T anotherObject) {  
    System.out.println(genObject + ", " + anotherObject);  
}
```

```
GenericObject<String> obj1 = new GenericObject ("Hi");
```

```
obj1.doSomething (new Integer (5));  
obj1.doSomething (new Animal ("Peter", 2));
```

Output:
Hi, 5
Hi, Name: Peter Age: 2

Q4 Bounded types

It is also possible to say that the type should exist in some hierarchy

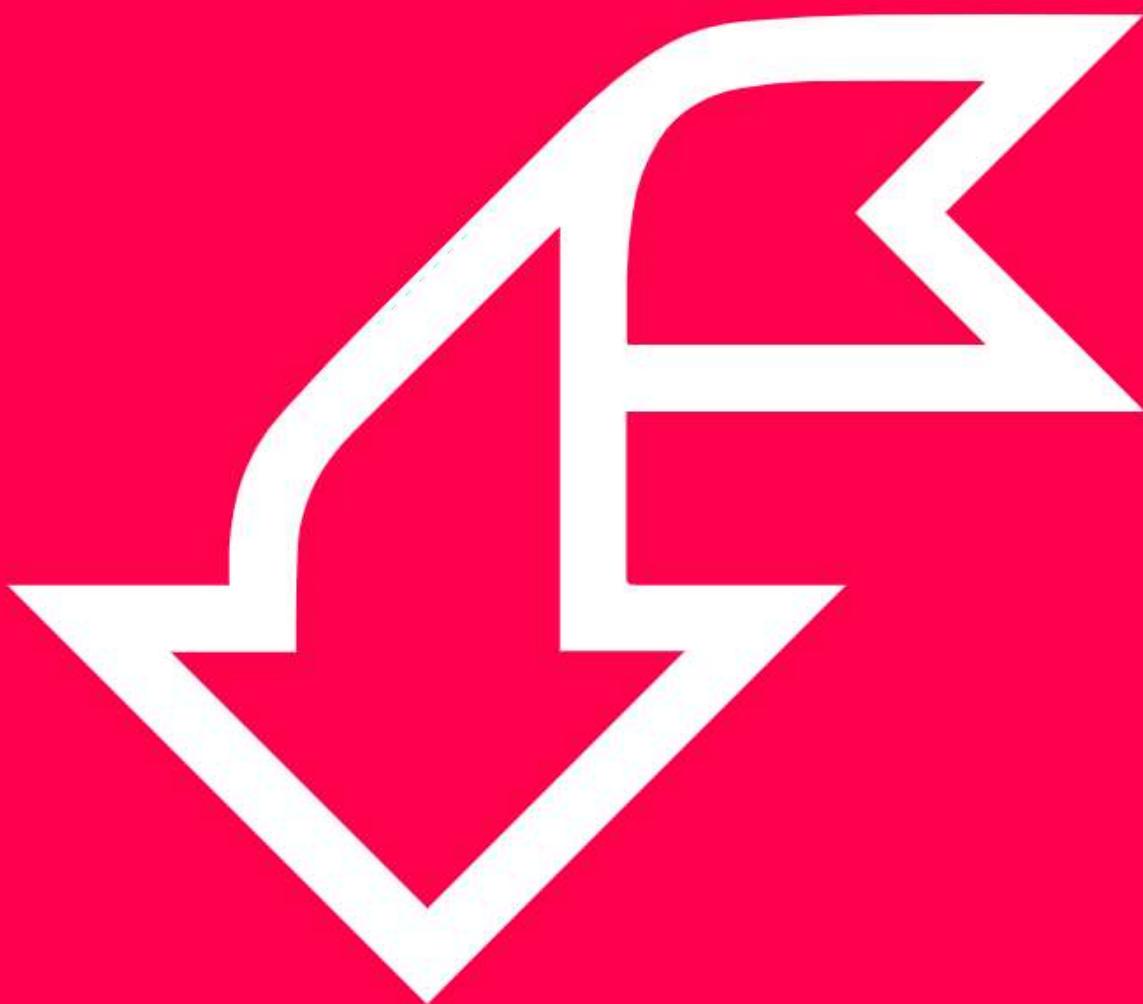
- For example: Allow any type in this class as long as it is an Animal or subtype of Animal

```
public class AnimalGeneric<T extends Animal> {  
    ...  
}  
  
AnimalGeneric<Rabbit> an1 = new AnimalGeneric<Rabbit>  
    (new Rabbit ("Peter", 2, true));  
AnimalGeneric<Animal> an2 = new AnimalGeneric<Animal>  
    (new Rabbit ("Benji", 3, true));  
// But this causes a Bound Mismatch error:  
AnimalGeneric<Book> an3 = new AnimalGeneric<Book>  
    (new Book ("Beyond Black", new String[5], 9.99));
```

CONVENTIONS

Generic types use a capital letter

- **E** - Element (used extensively by the Java Collections Framework)
- **K** - Key
- **N** - Number
- **T** - Type
- **V** - Value
- **S,U,... etc.** - 2nd, 3rd,... types



Exercise

**Looking at collections and generics in
more detail**

- Create collections with generics
- Iterate through collections
- Sorting collections

SUMMARY

Basic collection classes

- Lists
- Maps
- Sets

Using collections

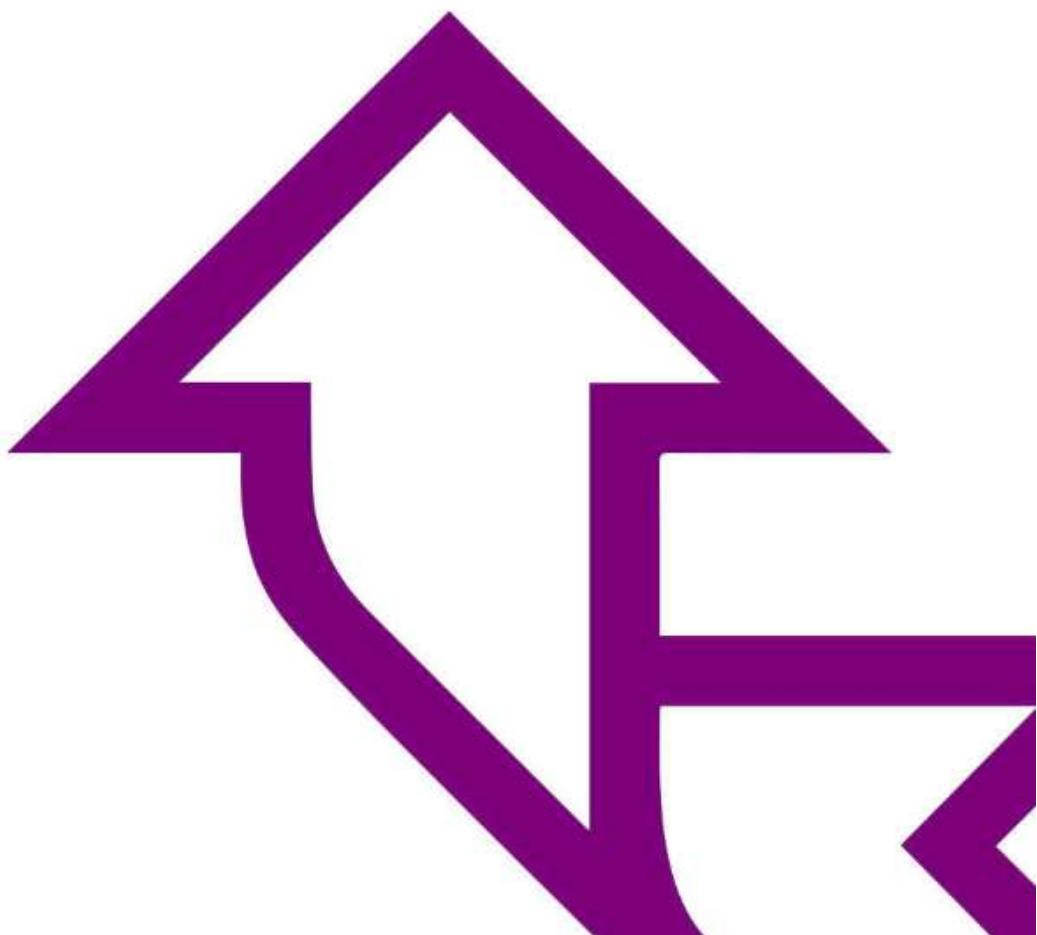
- Iterating over collections
- Sorting
- Comparators

Generics

- Bounded types



Exception Handling



OUTLINE

Error handling in Java

- Debugger

Exceptions

- What is an exception?
- Unchecked vs. checked

Handling exceptions

- Try ... Catch
- Finally
- Throws

Writing our own exceptions



OBJECTIVES

By the end of this session we should be able to:

- Understand and use exceptions in your code
- Use the try, catch, finally blocks
- Use the try-with-resources construct
- Write our own exceptions
- Use some basic read / write console operations

Qn Program errors

Many errors can occur when programming

- Syntax errors are picked up by IDEs and the compiler
- Logical errors are harder to identify

Some languages return error codes to indicate a problem

- Difficult to see what was going on in the code
- Needed to look up what the code meant

QA The IntelliJ Debugger

- Debuggers allow us to step through code, seeing the state of the program after each instruction
- This can help identify the error
- We can't ship a program that requires the user to do this!

The screenshot shows the IntelliJ IDEA interface with the following details:

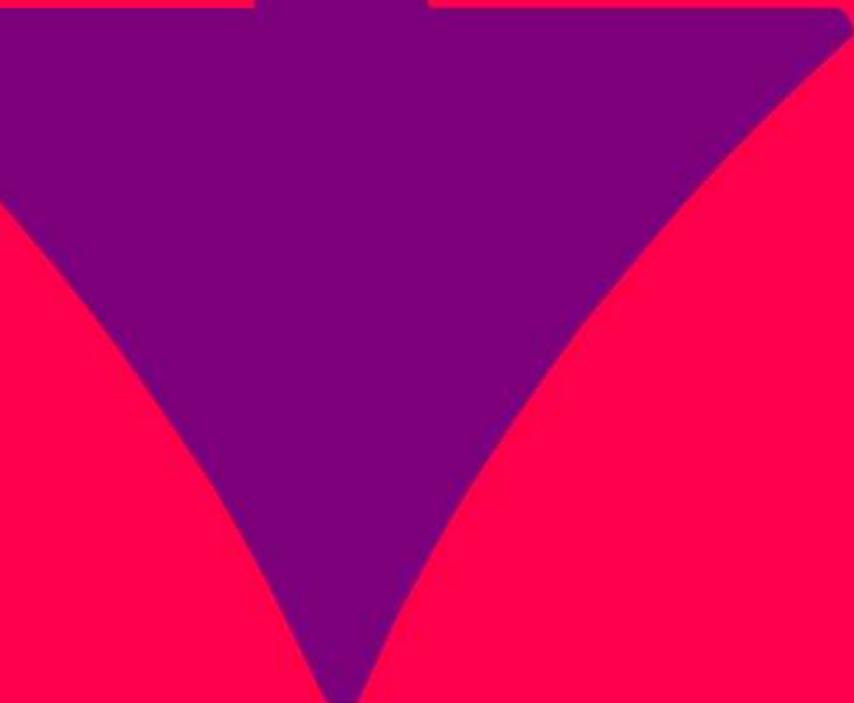
- Title Bar:** Shows "C_loops" and "C_loops.java".
- Code Editor:** Displays the following Java code:

```
javatut@Javatut-MacBook-Pro:~/Desktop$ C_loops.java
46      while(b < 10) {
47          b = 0;
48          while(c < 10) {
49              c = 0;
50              System.out.println(" " + b + " , " + c + ")");
51          }
52          c++;
53      }
54 }
```
- Breakpoints:** A red circular icon with a white dot is visible at line 46.
- Variable Watch Window:** Located at the bottom left, it shows:

args	{String[0]@733}[]
i	10
j	20
k	11
b	0
c	0
- Toolbar:** Includes icons for back, forward, search, and other navigation functions.

We need a way to recover from program failure gracefully and continue execution if possible

EXCEPTIONS

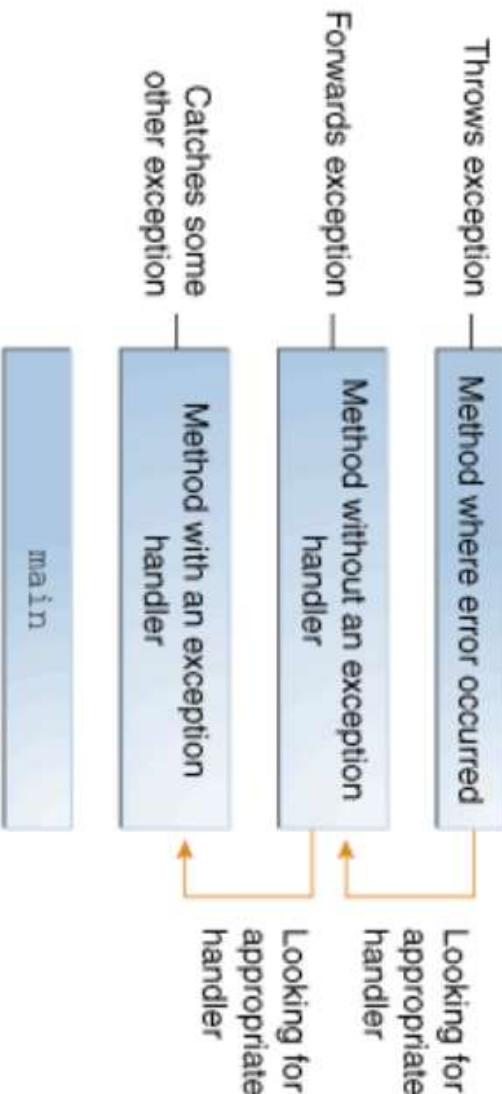


QA Exceptions

An exception is something that occurs during the execution of a program that disrupts the normal flow of execution

Exceptions are thrown by methods when they reach an error state they cannot recover from themselves

- When an error occurs an Exception object is created by the run time and passed back up the call stack until there is a method that can handle it, or the program stops entirely



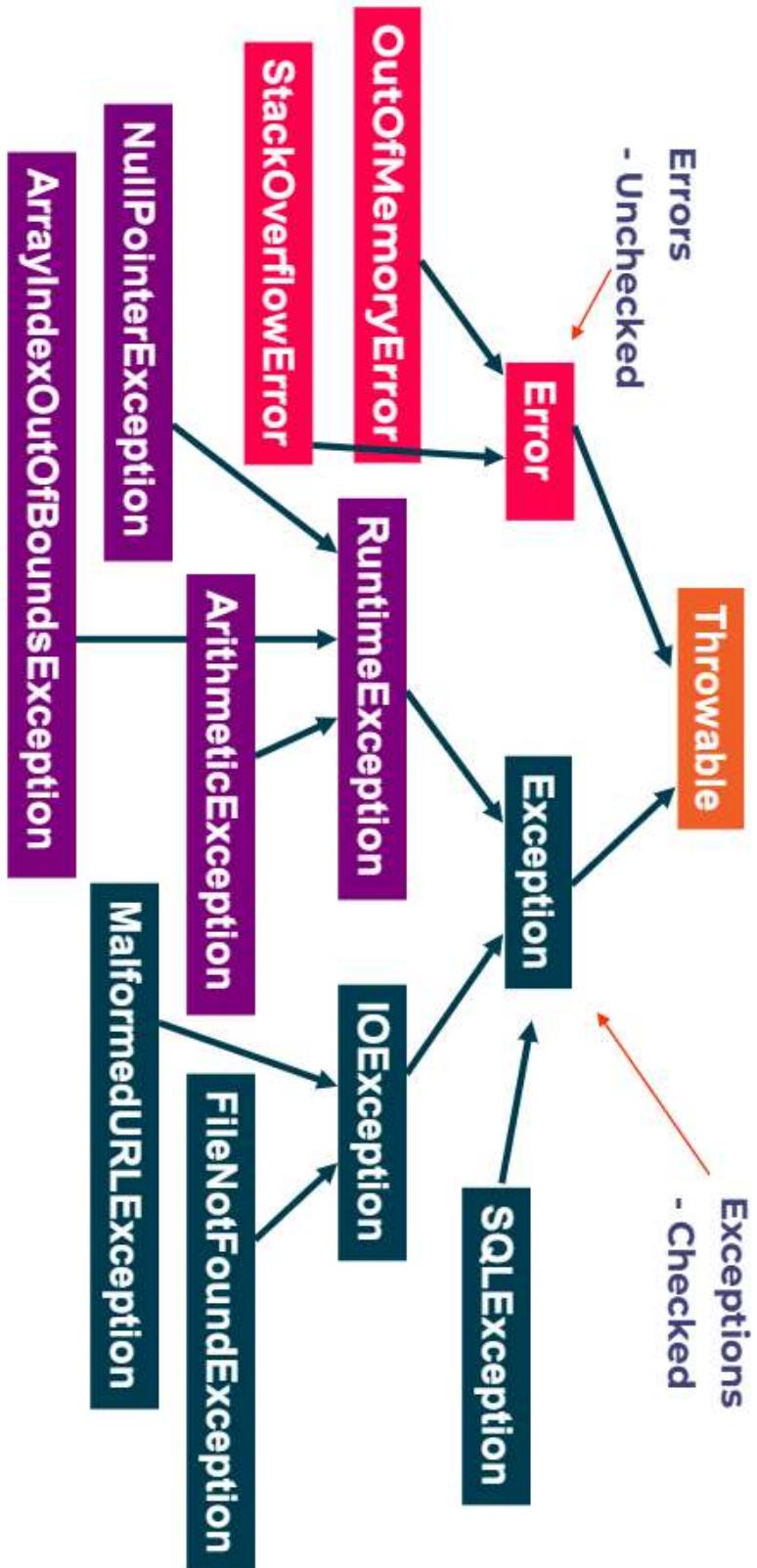
Qn Types of built in Exceptions

Parent object is of type Throwable

- There are two sub-classes, errors and exceptions
 - Errors are serious problems that should not be handled but should terminate the program
 - These are known as unchecked exceptions
- There are many Exception classes for different situations
- <http://docs.oracle.com/javase/17/docs/api/java/lang/Exception.html>
 - Exceptions contain a message that says what problem occurred during execution

getMessage()	Returns the message stored in the exception
getLocalisedMessage()	Localised version of the message
getStackTrace()	Print out the stack trace showing where the error occurred
toString()	Description of the object

QA Throwables Subclasses



EXCEPTIONS HANDLING

Q4 Handling exceptions

It can be important for a program to continue running even after an exception has happened

- Roll back changes to a database
- Close files
- Allow the user to carry on working even if there has been a problem

Java uses a try / catch block to handle exceptions

- Allows the program to 'catch' exceptions that have been thrown and recover gracefully
- You do not need to catch all exceptions
- Checked exceptions need to be caught and dealt with, runtime exceptions do not (although it is a good idea to)

Q4 try... catch

The basic form of the try/catch block uses the following format:

```
try {  
    //... some code ...  
} catch (SomeException e) {  
    //... handle the exception ...  
} catch (AnotherException e1) {  
    //... handle this exception ...  
}
```

- It is not best practice to catch all exceptions using the Exception class
 - you want to be as specific as possible when coding

Qn try... catch example

```
public static void main(String[] args) {  
    int[] array = new int[10];  
    array[10] = 42;  
    System.out.println("Continue to here");  
}
```

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException:  
at Main.main(Main.java:9)
```

: Index 10 out of bounds for length 10

Q/ try... catch example

```
public static void main(String[] args) {  
    int[] array = new int[10];  
  
    try {  
        array[10] = 42;  
    } catch (ArrayIndexOutOfBoundsException e) {  
        System.out.println(e);  
    }  
  
    System.out.println("Continue to here");  
}
```

java.lang.ArrayIndexOutOfBoundsException: Index 10 out of bounds for length 10
Continue to here

Q& finally

- There are some operations that we want to ensure always run, even if an exception is thrown, for this we use the **finally** block
- Most common use is to close resources that we've opened
 - When execution leaves try block after completing normally or exiting due to return or break
 - Or after an exception is thrown
- ... control is always passed to finally handler

```
BufferedReader br = null;  
String next;  
try  
{ br = new BufferedReader(new InputStreamReader(System.in));  
... // use reader  
} catch (...) {...} // catch possible Exceptions  
finally  
{ if (br != null){  
    br.close(); // will need its own try/catch  
}  
}
```

Q1 Putting it all together

```
public static void main(String[] args) {  
    BufferedReader br = null;  
try {  
    br = new BufferedReader(new InputStreamReader(System.in));  
    System.out.println("Enter a line of text");  
    String line = br.readLine();  
    while (!line.equals("stop")) {  
        System.out.println(line);  
        System.out.println("Enter a new line of text");  
        line = br.readLine();  
    }  
}catch (IOException e) {  
    e.printStackTrace();  
}finally {  
    /*will also need a try/catch block  
     * if (br != null) { br.close(); }  
    }  
}
```

QA Enhanced try

The try block now has syntax to define resources

- Resources are automatically closed on exit of try

```
try (BufferedReader br = new BufferedReader(Source))  
{  
    // conventional I/O read code  
}
```

Could define more than one resource, using ";" to separate them

```
try ( BufferedReader br = new BufferedReader  
      (new InputStreamReader(System.in));  
      BufferedWriter bw = new BufferedWriter  
      (new OutputStreamWriter(System.out));  
 )  
{  
    // conventional I/O read/write code  
}
```

Q4 Enhanced catch

The catch header can define/catch alternatives

- Overloaded use of the '||' operator

```
catch ( ArrayIndexOutOfBoundsException |  
       NumberFormatException ex  
     )  
{ // Code to handle both types }
```

The catch block will rethrow the actual IOException subclass:

```
catch (final IOException e)  
{  
    ...  
    throw e;  
}
```

Can mix and match with conventional catch clauses

- Polymorphic order still established

QA IntelliJ Help

IntelliJ will show errors if exceptions are not handled

- It can automatically surround the code in a try/catch block or add a catch to an existing try
- Includes the specific exceptions that are being thrown

```
1 package com.qa;
2
3 import java.io.BufferedReader;
4 import java.io.InputStreamReader;
5
6 public class ReadInput {
7
8     public static void main(String[] args) {
9         // TODO Auto-generated method stub
10
11     BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
12
13     System.out.println("Enter a line of text");
14
15     String line = br.readLine();
16
17 }
18
19 }
```

💡 Add exception to method signature >

💡 Surround with try/catch

QA Throwing exceptions

We can see if an exception is thrown by an API method by looking at the JavaDoc

- Uses the @throws annotation

readLine

```
public String readLine()  
    throws IOException
```

Reads a line of text. A line is considered to be terminated by any one of a line feed ('\n'), a carriage return ('\r'), or a carriage return followed immediately by a linefeed.

Returns:

A String containing the contents of the line, not including any line-termination characters, or null if the end of the stream has been reached

Throws:

IOException - If an I/O error occurs

See Also:

`Files.readAllLines(java.nio.file.Path, java.nio.charset.Charset)`

Q1 Throwing Exceptions

Sometimes we don't want a method to deal with an exception itself

- We want to throw the exception back to the calling method to handle
- Allows the method to handle the logic rather than logic and exception

```
public void readInputLines() throws IOException {
    try (BufferedReader br = new BufferedReader(
        new InputStreamReader(System.in))) {
        System.out.println("Enter a line of text");
        String line = br.readLine();
        while (!line.equals(" stop ")) {
            System.out.println(line);
            System.out.println("Enter a new line of text");
            line = br.readLine();
        }
    }
}
```

WRITING YOUR OWN EXCEPTIONS

Qn Writing our own exceptions

To write an exception of our own we extend the Exception class

```
public class BadLineException extends Exception {  
    String line;  
    public BadLineException() {  
        super("I didn't like the line I read in");  
    }  
    public BadLineException(String msg, String line) {  
        super(msg);  
        this.line = line;  
    }  
    public String getBadLine() {  
        return line;  
    }  
  
    if (line.equals("no")) {  
        throw new BadLineException("Line said no", line);  
    }  
}
```

SUMMARY

Error handling in Java

- Debugger

Exceptions

- What is an exception?
- Unchecked vs. checked

Handling exceptions

- Try ... Catch
- Finally
- Throws

Writing our own exceptions



Exercise

Using and writing exceptions in Java

- Use console I/O operations to read and write to the console
- Use the try/catch block to recover from any errors in the code
- Use the try with resources construct to automatically close open resources



File Handling



OUTLINE

File handling in Java

- I/O streams
- Stream classes
- Types of stream classes
- Byte stream classes vs. character stream classes

Read and write operations

- Reading from a file
- Writing to a file



QA

OBJECTIVES

By the end of this session we should be able to:

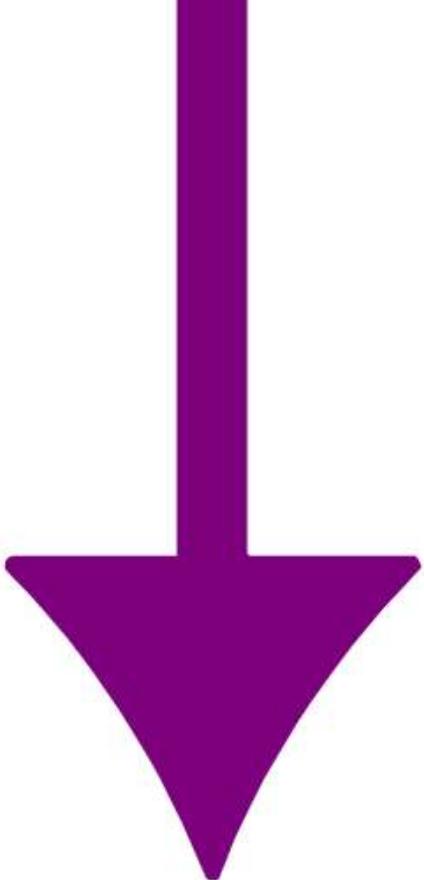
- Understand and use Java I/O Stream classes
- Use basic read / write file operations

I/O Streams

I/O Streams allow our Java programs to communicate with I/O devices and files

Java programming uses two type of streams -
input streams and output streams

Input streams are for reading activities whereas
output streams are for writing



Q4 Stream classes

Java I/O Stream classes are defined in the java.io and java.nio packages

Java's I/O Stream classes are classified as two types

- Byte streams
- Character streams

Byte stream classes

- Handle raw binary data - do not support the UNICODE character set
- InputStream and OutputStream

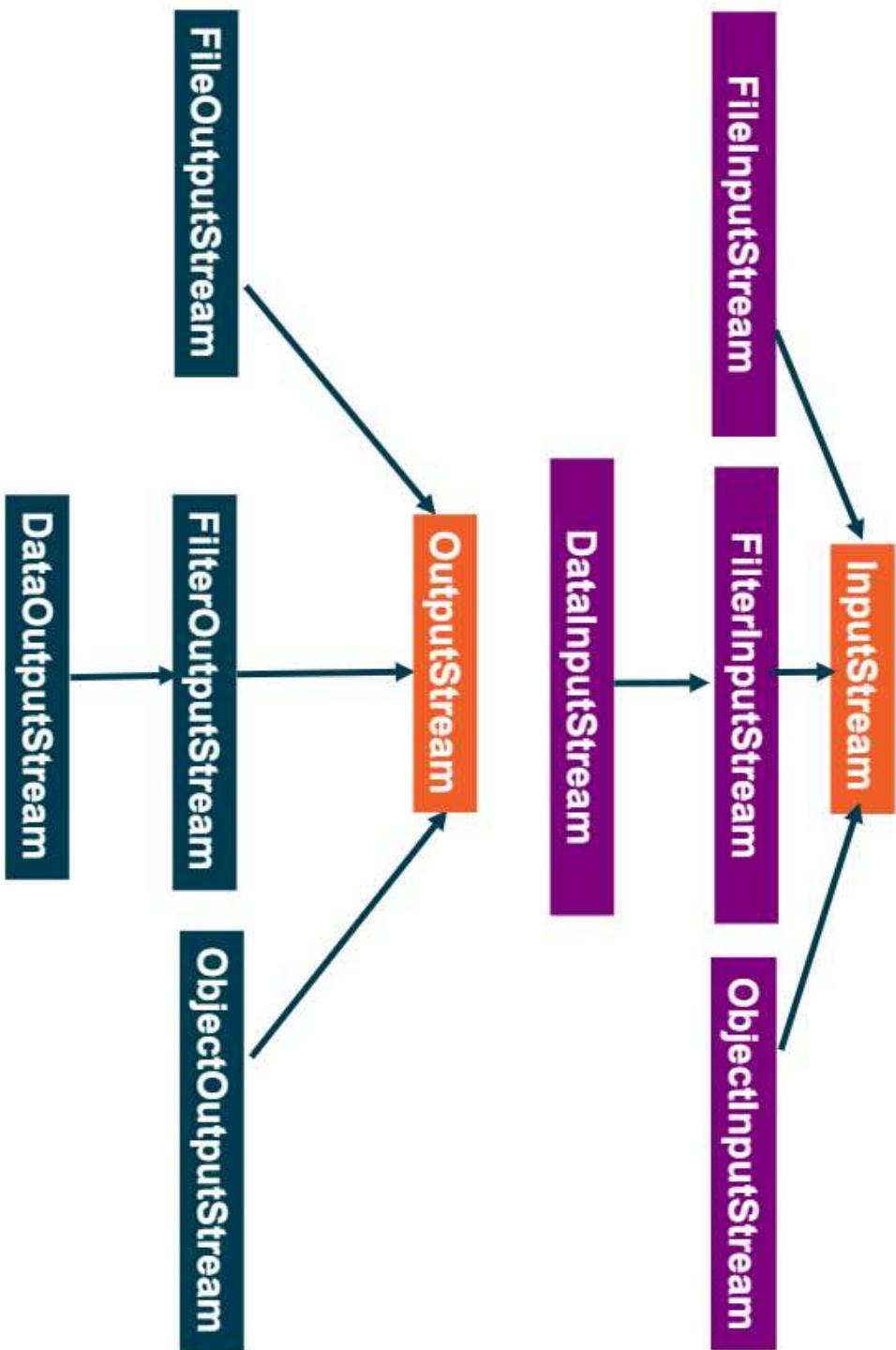
Abstract classes - have their own subclasses

Character stream classes

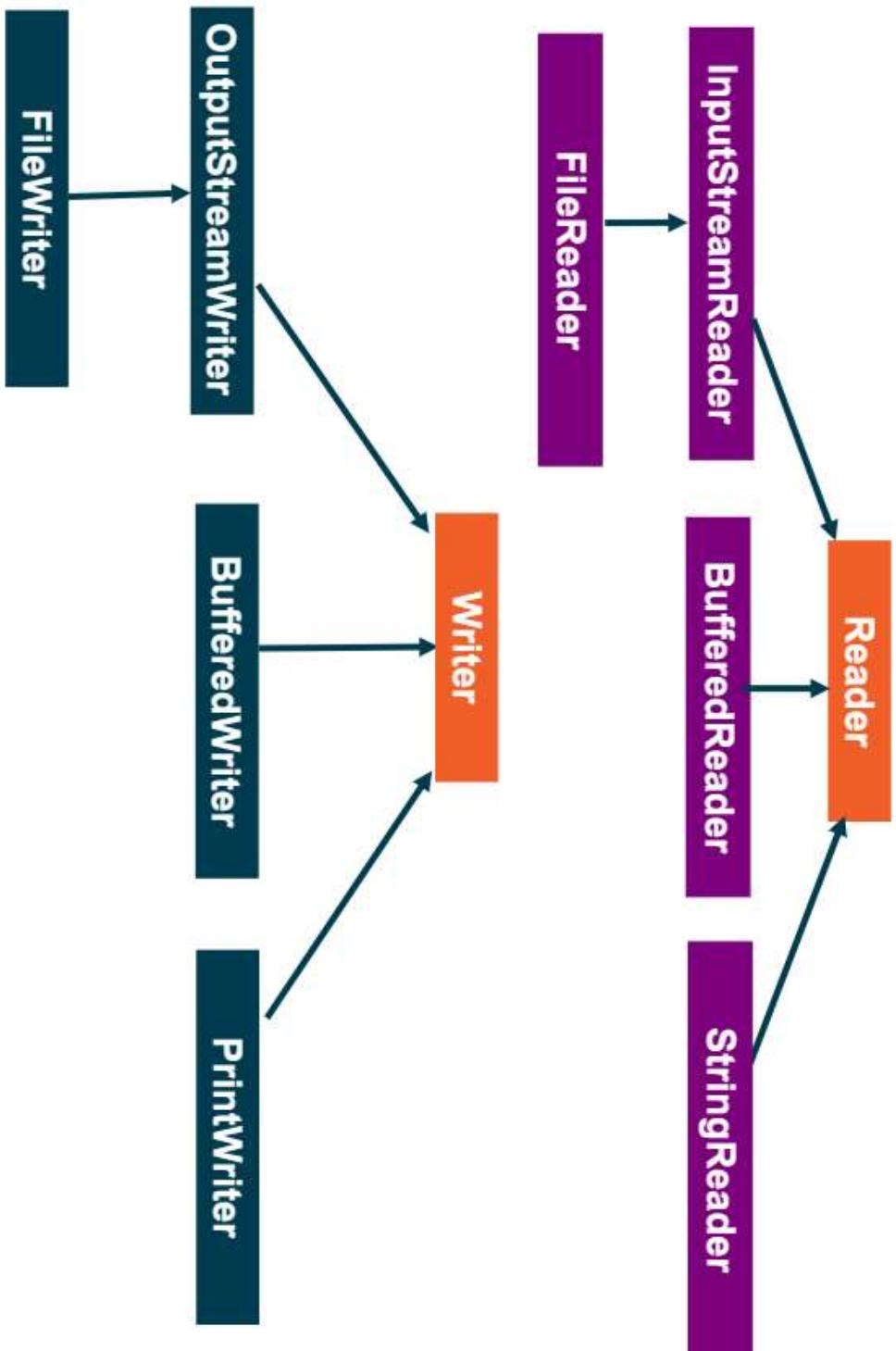
- Handle I/O of character data - support UNICODE characters
- Reader and Writer

Abstract classes - have their own subclasses

Q4 Byte stream classes



Q4 Character stream classes





Qn BufferedReader, InputStreamReader, and FileInputStream

```
BufferedReader br = null;
FileInputStream is = null;
String next;
try
{
    is = new FileInputStream("input.txt");
    br = new BufferedReader(new InputStreamReader(is));
    ...
    // use buffered reader
} // catch Exceptions here
finally
{
    // would need further Exception handling
    if (br!= null)
        br.close();
    if(is!=null)
        is.close();
}
```

Bridge between
BufferedReader and
FileInputStream

Q4 BufferedReader and FileReader

```
BufferedReader in = null;  
String next;  
try  
{  
    in = new BufferedReader(new FileReader(file));  
    ... // use reader  
}  
finally  
{ // would need further exception handling  
    if (in != null)  
        in.close();  
}
```

No bridge
needed

QA IntelliJ Help with I/O exceptions

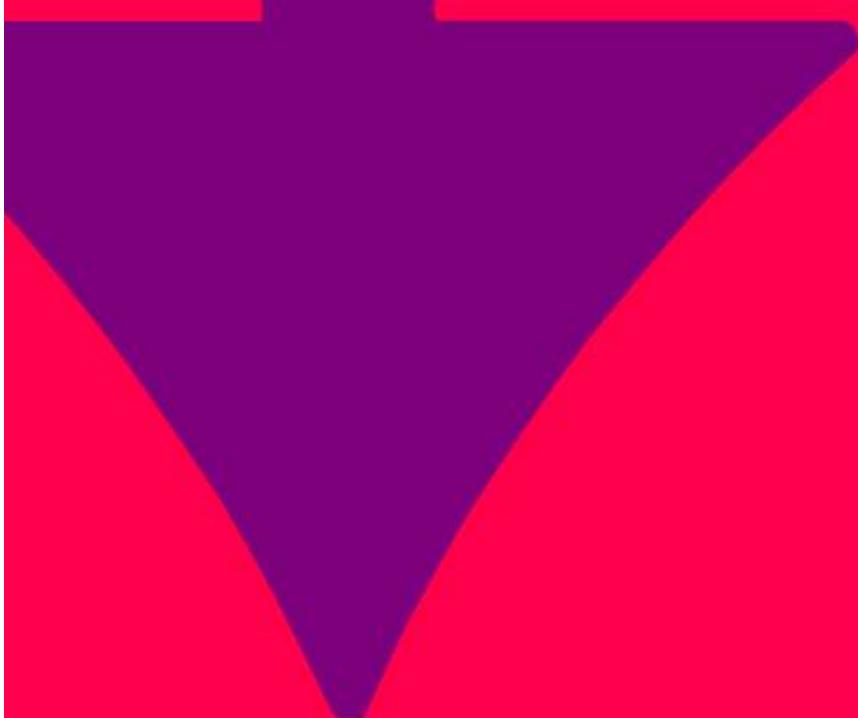
IntelliJ will show errors if I/O exceptions are not handled

- It can automatically surround the code in a try/catch block or add a catch to an existing try
- Includes the specific exceptions that are being thrown

```
1 package com.qa;
2
3 import java.io.BufferedReader;
4 import java.io.InputStreamReader;
5
6
7 public class ReadInput {
8     public static void main(String[] args){
9         BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
10        System.out.println("Enter your text: ");
11        String textLine = br.readLine();
12        System.out.println("You entered: " + textLine);
13    }
}
```

① Add exception to method signature > ② Surround with try/catch

READ AND WRITE OPERATIONS



Q4 Reading from a file

```
public static void main(String[] args) {  
    BufferedReader br = null;  
    try {  
        br = new BufferedReader(new FileReader("input.txt"));  
        String line = br.readLine();  
        while (line != null){  
            System.out.println(line);  
            line = br.readLine();  
        }  
    } catch (FileNotFoundException e) {  
        e.printStackTrace();  
    } catch (IOException e) {  
        e.printStackTrace();  
    } finally {  
        //will also need a try/catch block  
        if (br != null) br.close();  
    }  
}
```

Q1 Writing console input to a file

```
public static void main(String[] args) {  
    BufferedWriter bw = null; BufferedReader br = null;  
    try{bw = new BufferedWriter(new FileWriter("output.txt"));  
     br = new BufferedReader(new InputStreamReader(System.in));  
     System.out.println("Enter a line of text");  
     String line = br.readLine();  
     while (!line.equals("stop")) {  
         bw.write(line + "\n"); // Writes to output.txt  
         System.out.println("Enter a line or STOP to quit");  
         line = br.readLine(); // Reads from console  
     }  
     bw.flush(); // required  
    } catch (FileNotFoundException e) {  
        e.printStackTrace();  
    } catch (IOException e) {  
        e.printStackTrace();  
    } finally{}  
}
```

Qn Throwing exceptions

Sometimes we don't want a method to deal with an exception

- We want to throw the exception back to the calling method to handle
- Allows the method to focus on the logic rather than logic and exception handling

```
public void readFile() throws FileNotFoundException, IOException{
    BufferedReader br = null;
    try {
        br = new BufferedReader(
            new FileReader(new File("input.txt")));
        String line = br.readLine();
        while (line != null) {
            System.out.println(line);
            line = br.readLine();
        }
    } finally {
        if (br != null) br.close();
    }
}
```

QA Try-with-resources

The try-with-resources construct allows us to define resources on entry to the try block

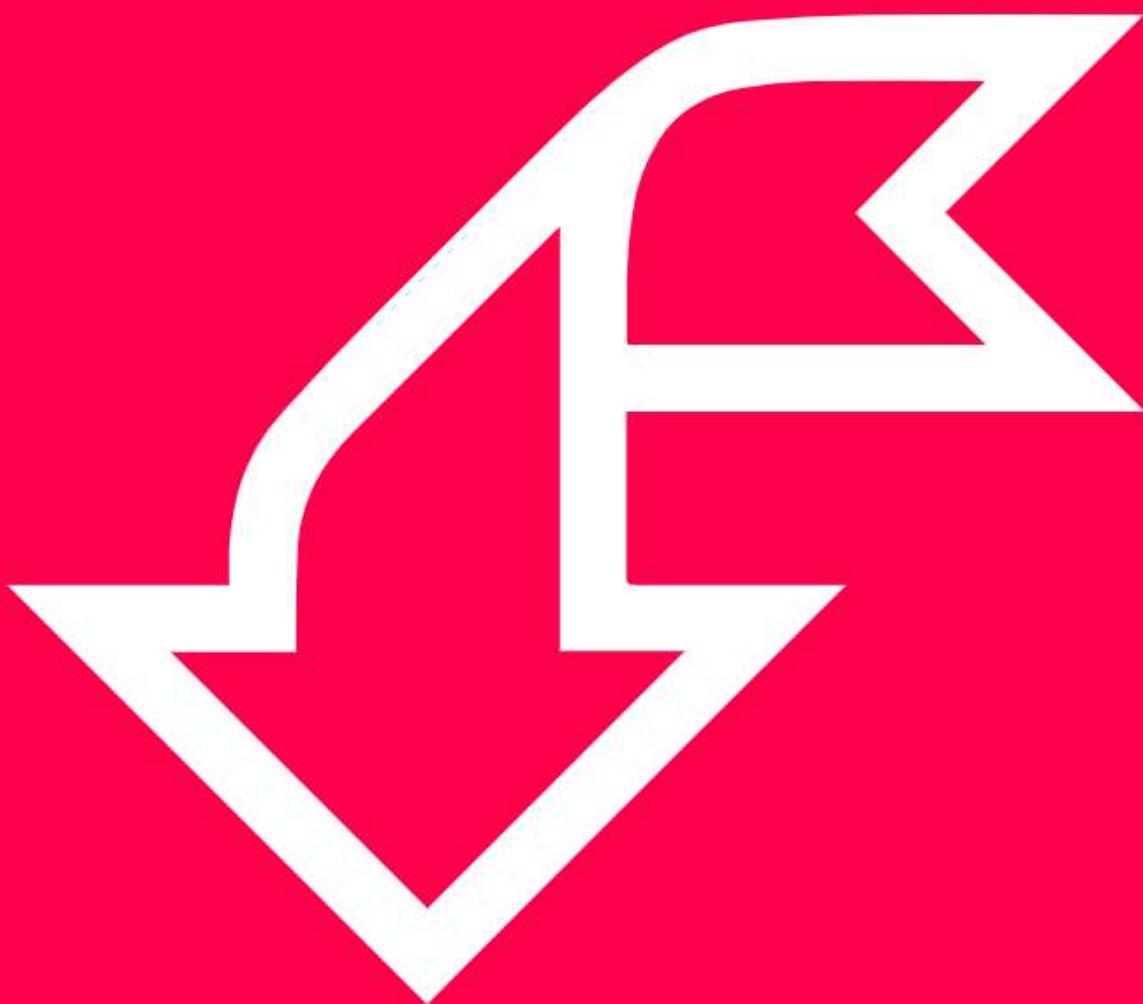
- Resources are automatically closed on exit of try

```
try( BufferedReader in = new BufferedReader(new FileReader("input.txt")))
{
    // process the input lines
}
```

Can define more than one resource

- Use ; separated statements

```
try(
    BufferedReader in = new BufferedReader(new FileReader("input.txt"));
    BufferedWriter out = new BufferedWriter(new FileWriter("output.txt"));
)
{
    // copy selected input lines to the output file
}
```



Exercise

Using and writing exceptions in Java

- Use try-with-resources and file I/O operations to read and write to files
- Use the try/catch block to recover from any errors in the code
- Recall exception handling and write your own exception
- Throw an Exception to the calling method

SUMMARY

File handling in Java

- I/O streams
- Stream classes
- Types of stream classes?
- Byte stream classes vs. character stream classes

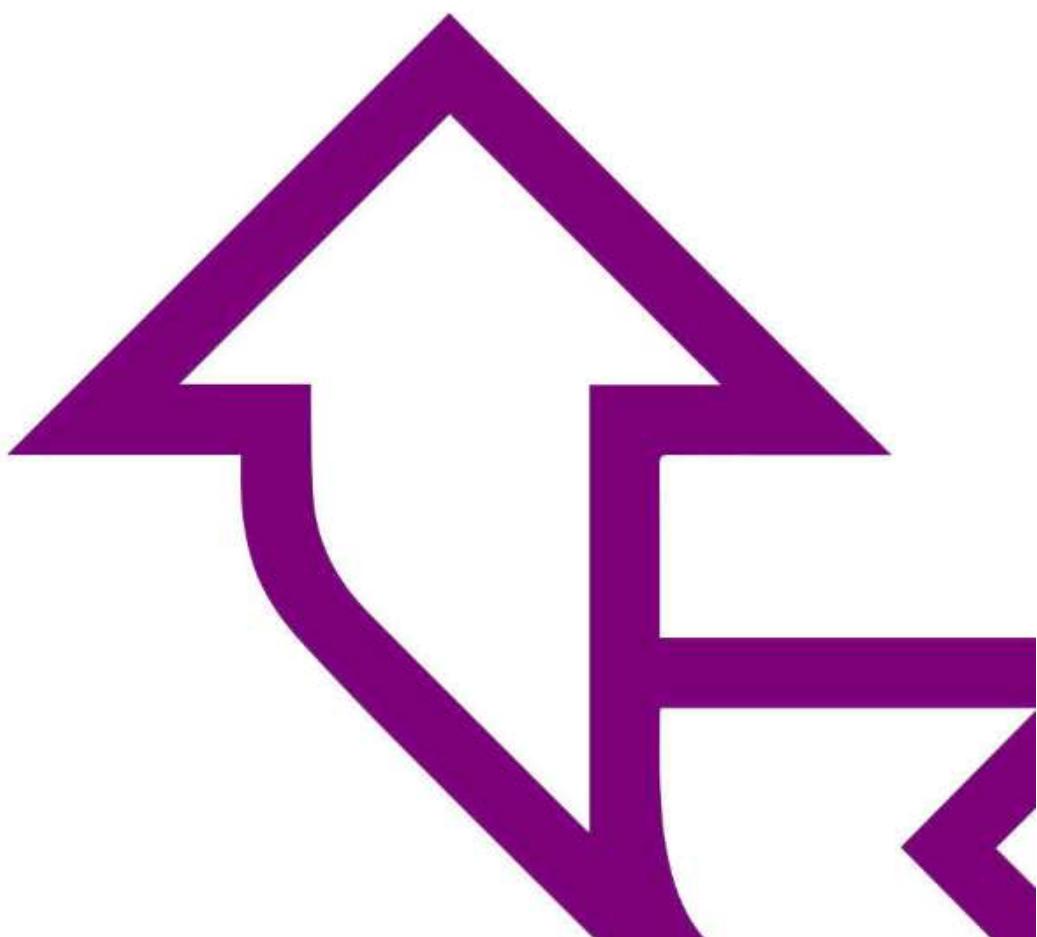
Read and write operations

- Reading from a file
- Writing to a file





String Handling

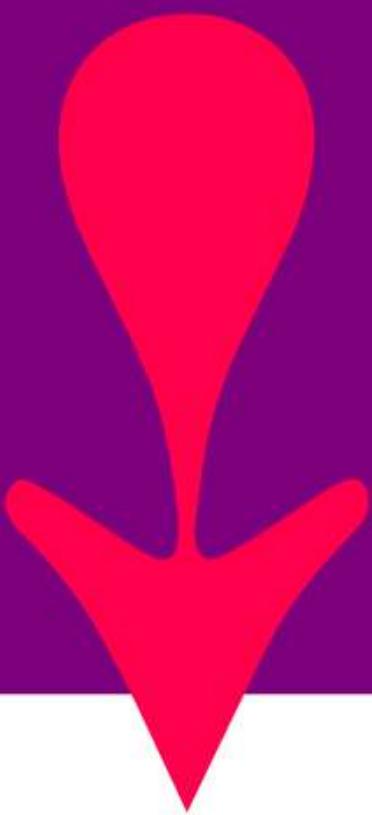


OUTLINE

The string class

- What we've already used – literals, equals()
- Altering case and locating characters
- Substringing and splitting
- String comparisons
 - matches() and simple regular expressions

JDK API documentation



OBJECTIVES

By the end of this session, we should be able to:

- use some of the String class manipulation methods
- use the JDK API documentation to investigate details of a class

Q1 What we've already used

- Literals, concatenation, equality testing

```
String hello = "Hello ";
String world = "World";
String greeting = hello + world;
System.out.println(greeting);
String start = "My age is ";
int age = 21;

String complete = start + age;
System.out.println(complete);

System.out.println(hello.equals(world));
String world2 = "world";
System.out.println(world.equalsIgnoreCase(world2));
```

Qn Altering case and locating characters

- `toLowerCase()`, `toUpperCase()`, `indexOf()`, `charAt()`

```
String title = "Introduction to Programming with Java";  
System.out.println(title);  
System.out.println(title.toLowerCase());  
System.out.println(title.toUpperCase());  
  
System.out.println(title.indexOf("Java")); // 33  
System.out.println(title.charAt(16)); // 'P'  
  
System.out.println(title.charAt(title.indexOf("Java"))); // 'J'
```

Qn Substringing and splitting

- length(), substring(), split()

```
String title = "Introduction to Programming with Java";           // 37
System.out.println(title.length());                                 // "Introduction"
// return String from start char index up to but not including end char index
System.out.println(title.substring(0, 12));                         // "Introduction"
System.out.println(title.substring(33, title.length()));             // "Java"
System.out.println(title.substring(33));                            // "Java"
System.out.println(title.substring(title.indexOf("Java")));        // "Java"

// retrieve Strings separated by a specific value
String[] words = title.split(" ");
for(String word: words) {
    System.out.println(word);
}
// if you just want the word at
// index 2 (3rd word):
System.out.println(title.split(" ")[2]);
```

Qn String comparisons

- `startsWith(), endsWith(), contains()`

```
String title = "Introduction to Programming with Java";  
  
if (title.startsWith("Introduction")) {  
    System.out.println(title + " is an entry level course");  
}  
  
System.out.println("Is it a Java course? " + title.endsWith("Java"));  
  
if (title.contains("Programming")) {  
    System.out.println("It looks at aspects of Programming");  
}
```

Qn String comparisons

- `compareTo()`

```
String title = "Introduction to Programming with Java";
String[] langs = {"Java", "Ruby", "Python", "C#", "C++", "C", "SQL"};
for(String lang: langs) {
    System.out.println(lang);
}

// can be used to test for 1 string > another
System.out.println(langs[0].compareTo(langs[1])); // -ve (Java < Ruby)
System.out.println(langs[1].compareTo(langs[2])); // +ve (Ruby > Python)
System.out.println(langs[0].compareTo(title.substring(33))); // 0 (equal)

// primarily used for sorting
Arrays.sort(langs);
for(String lang: langs) {
    System.out.println(lang);
}
```

QA Pattern matching: regular expressions

- Regular expressions allow patterns rather than exact matches to be found
- Use a set of metacharacters (special meanings) and literals
- Some of the commonly used metacharacters:

Example	Matches
a.b	a followed by any character followed by b
ab+c	a followed by 1 or more bs followed by c
ab?c	a followed by 0 or 1 bs followed by c
ab*c	a followed by 0 or more bs followed by c
a.*b	a followed by any number of any characters, then b
ab{3}c	a followed by exactly 3 bs followed by c
ab{3,}c	a followed by at least 3 bs followed by c
ab{3, 5}c	a followed by between 3 and 5 bs followed by c
a[bcd]e	a followed by b or c or d followed by e
Ste(v ph)en	Ste followed by v or ph followed by en
Happy\?	Happy? – treats ? as a literal

Qn Pattern matching: regular expressions

- Several String methods use regular expressions, including matches()

```
for(String name: names) {  
    if (name.matches(".*[io]")) {  
        System.out.println(name + " ends with i or o");  
    }  
    if (name.matches(".*o.*")) {  
        System.out.println(name + " has o in the 3rd character");  
    }  
    if (name.matches(".*it?a")) {  
        System.out.println(name + " ends with ita or ia");  
    }  
    if (name.matches("T.*a")) {  
        System.out.println(name + " starts with T and ends with a");  
    }  
    if (name.matches("H[aeo].*(l{1,}|r{1,})y")) {  
        System.out.println(name + " starts with H, then a, e or o, " +  
            " and ends with at least 1 letter l or r, then y");  
    }  
}
```

QA

JDK API DOCUMENTATION

QA JDK API documentation

- All String methods are fully explained in the API Docs

The screenshot shows the Java API Docs for the `String` class. The URL is <https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/String.html>. The page title is "String (Java SE 17 & JDK 17)". The navigation bar includes links for Overview, Module, Package, Class (highlighted), Use, Tree, Preview, New, Deprecated, Index, and Help. A search bar at the bottom right contains the placeholder "Search". The main content area displays the `String` class definition, its inheritance from `Object`, and its implementation of various interfaces. It highlights the `matches` method, which checks if a string matches a regular expression. The `Parameters:` section specifies the `regex` parameter, and the `Returns:` section states that it returns true if and only if the string matches the given regular expression.

String (Java SE 17 & JDK 17) x +

← → ⌂ ⌂ <https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/String.html>

Java API Docs (Java... Oracle Database 5... Oracle Database Da... Oracle Database 19... Oracle 19c License... Oracle Database Te... Oracle Cloud Infrastructure

OVERVIEW MODULE PACKAGE CLASS USE TREE PREVIEW NEW DEPRECATED INDEX HELP

SUMMARY NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

SEARCH: Search

Module java.base
Package java.lang

Class **String**

java.lang.Object
java.lang.String

matches

```
public boolean matches(String regex)
```

Tells whether or not this string matches the given regular expression.

All Implemented Interfaces:
Serializable, CharSequence, Comparable<String>, Constable, ConstString

public final class String
extends Object
implements Serializable, Comparable<String>, CharSequence, Constable, ConstString

The String class represents character strings. All string literals in Java®
Strings are constant; their values cannot be changed after they are created.
shared. For example:

```
String str = "abc";
```

Since: 1.4

SUMMARY

The string class

- What we've already used – literals, equals()
- Altering case and locating characters
- Substringing and splitting
- String comparisons
 - matches() and simple regular expressions

JDK API documentation

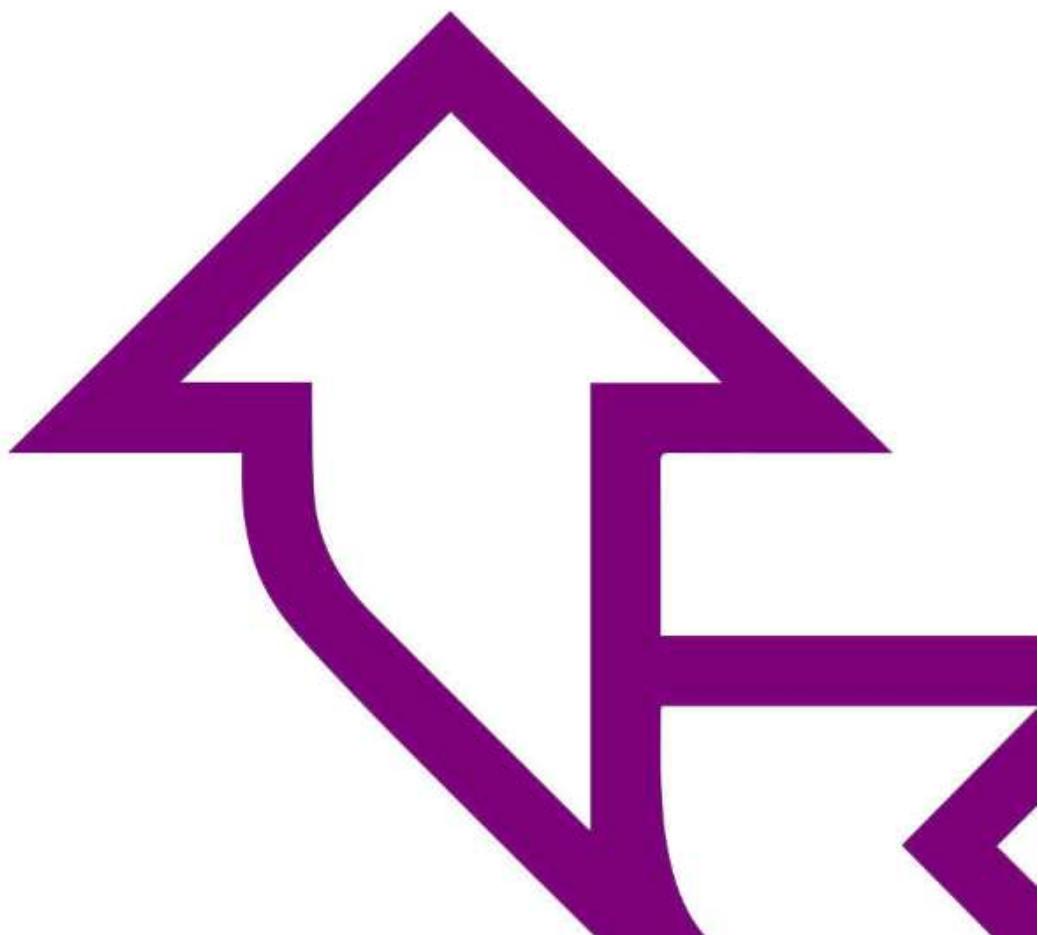


Exercise

- Use a variety of String methods to manipulate data
- Apply simple regular expressions in the context of the matches() method
- Experiment further with the methods as you use them
- Make use of the JDK API Documentation to explore further methods



Introduction to Functional Programming in Java



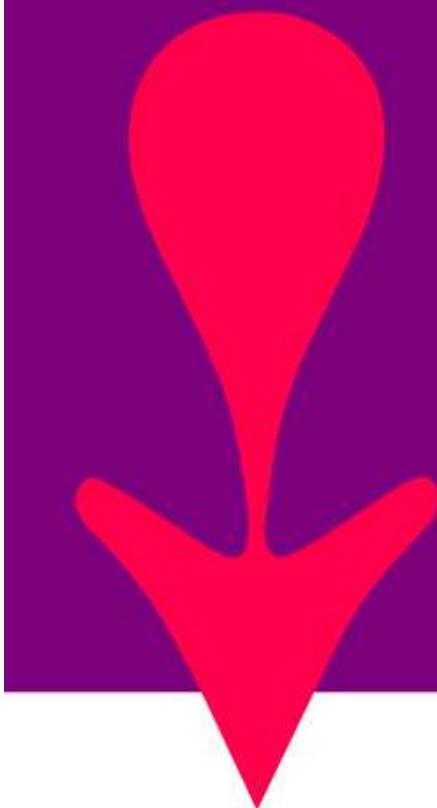
OUTLINE

What is functional programming?

Functional programming in Java

- Lambda functions
- Passing methods as arguments

Lambda arguments to collection methods



OBJECTIVES

By the end of this session we should:

- be able to identify Lambda expressions and use them with collections
- know how to work out the type of a Lambda

WHAT IS FUNCTIONAL PROGRAMMING?

Functional programming is a different paradigm to OO or Procedural programming:

- Has its roots in academia
 - Programs are constructed by applying and composing functions
 - Declarative rather than imperative
- Functions can be:
- Associated with names
 - Passed as arguments
 - Returned from other functions
- Functions are designed to:
- Always produce the same output if given the same input
 - Have no side effects
 - Not to store state

Qn Comparing Programming Paradigms

Functional programming

- Functions do not store state
- Functions can be composed
- No side effects
- Recursion focus
- Functions can be passed to methods

Object-oriented / procedural programming

- Objects store state and methods manipulate this
- Methods can call other methods
- Side effects happen due to state or exceptions
- Use loops (for/while)
- Classes are defined

QA Functional programming in Java

Functional programming in Java allows us to provide methods for functional interfaces as Lambda expressions.

Functional interface – one that has a single abstract method to be implemented

Lambda expressions (also known as anonymous functions)

- Enable us to pass methods as arguments to other methods
- Widely used by classes in the Collections framework
- For example, the **forEach()** method takes a Lambda expression that allows us to specify how each element of a List should be used or consumed:

```
list.forEach(s -> System.out.println(s));
```

Q1 Lambda expressions

Lambda expressions are like anonymous functions

► Three parts to the expression

Parameters	->	Body
s	->	System.out.println(s)
(int x, int y)	->	{ return x + y; }
()	->	{return "Hello World";}

- If there is more than one parameter then brackets () are needed
- If there is more than one statement in the body then braces {} are used
- The input parameters and return type can be inferred by the compiler

They can be used with other functional constructs or can be passed into methods

Q A The type of a Lambda

What type of parameter allows us to pass Lambda expressions?

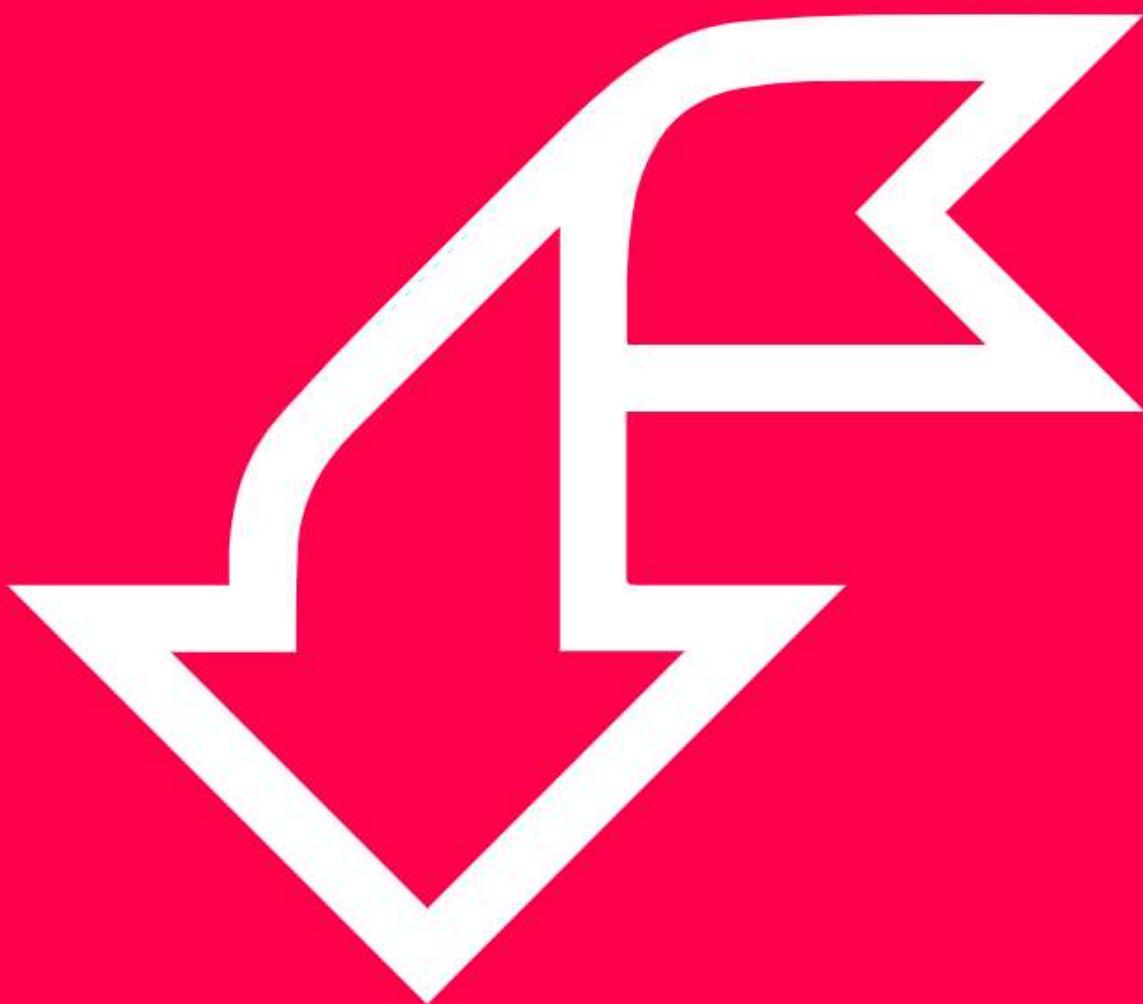
```
callMethod(list, s -> System.out.println(s));
```

The **java.util.function** package has a set of interfaces for many different types of Lambda expressions. You can also write your own functional interfaces.

```
private static void callMethod(  
    ArrayList<String> string,  
    Consumer<? super String> p) { ... }
```

Qn Lambda arguments to collection methods

```
ArrayList<String> names = new ArrayList<>();  
names.add("Rod"); ...  
  
names.replaceAll(s -> s.toUpperCase()); //Unary Operator  
System.out.println("Names changed to upper case");  
System.out.println(names);  
  
names.sort((s1, s2) -> s2.compareTo(s1)); //Comparator  
System.out.println("Names sorted in reverse");  
System.out.println(names);  
  
names.removeIf(s -> s.startsWith("R")); //Predicate  
System.out.println("Names without initial R");  
System.out.println(names);
```



Exercise

- Use a variety of ArrayList methods that have Lambda arguments

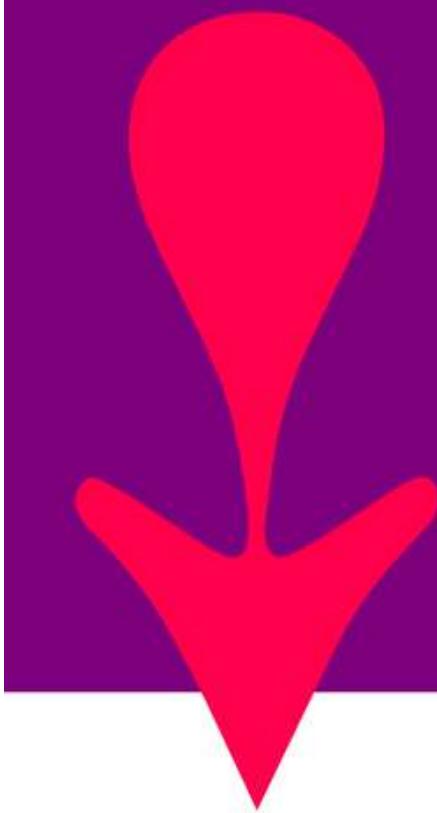
SUMMARY

What is functional programming?

Functional programming in Java

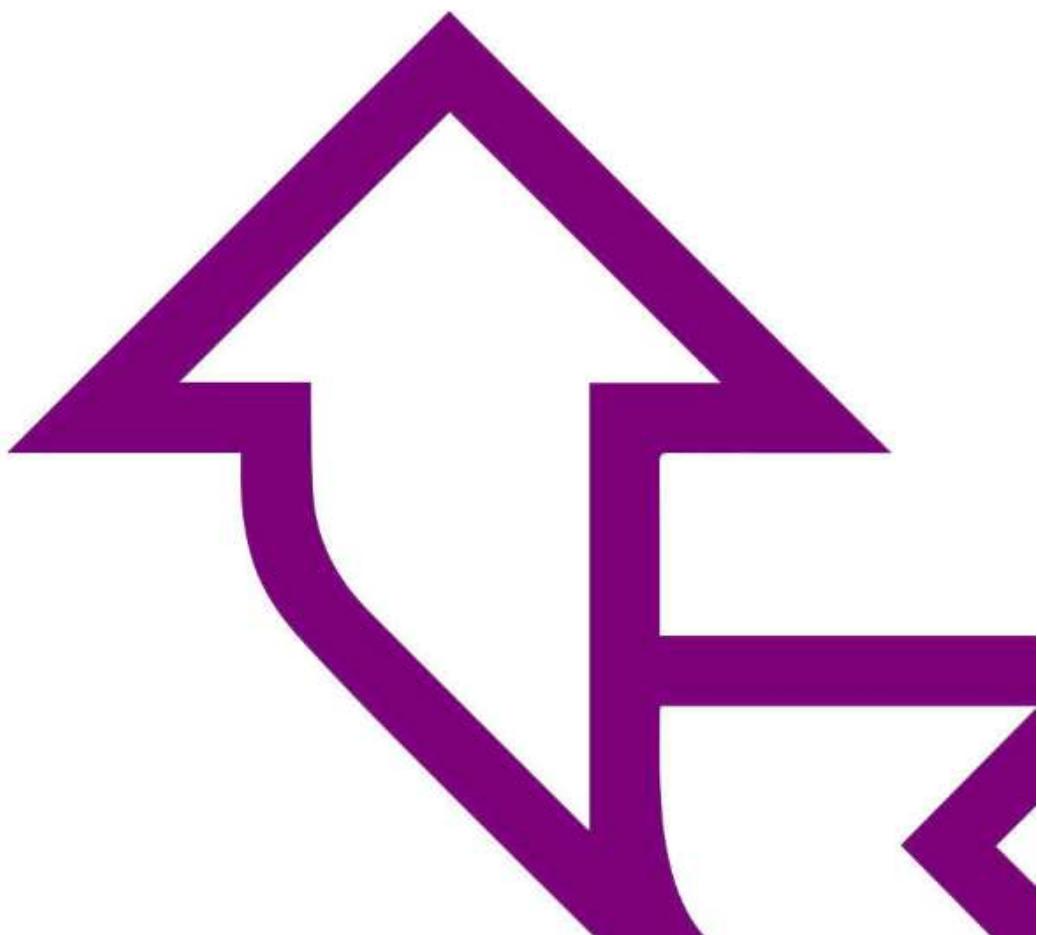
- Lambda functions
- Passing methods as arguments

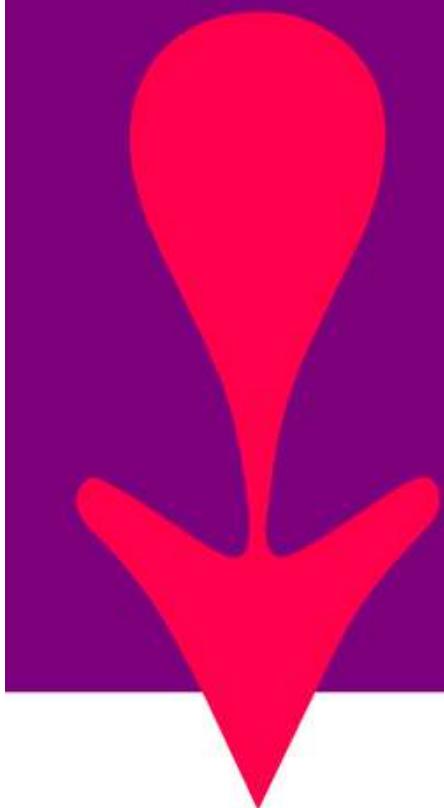
Lambda arguments to collection methods





Introduction To Streams





OUTLINE

Streams

- map, filter, collect, and reduce

Virtual extension methods

- What and why

QA

OBJECTIVES

By the end of this session we should:

- know how to chain operations together to form a pipeline
- be able to use default implementations in interfaces

Q1 The type of a Lambda

Recall that to help us write methods with Lambda parameters

```
callMethod(list, s -> System.out.println(s));
```

java.util.function has a set of functional interfaces that can be used, for example, the Consumer interface

```
private static void callMethod(  
    ArrayList<String> string,  
    Consumer<? super String> p) {
```

Q4 java.util.function basic function shapes

Consumer<T>
An operation that accepts a single input argument of type T and returns no result

Predicate<T>

A predicate (boolean-valued function) of one argument of type T

Supplier<T>

A supplier of results of type T

Function<T, R>

A function that accepts one argument of type T and produces a result of type R.

DoubleFunction<R>

A function that accepts a double-valued argument and produces a result of type R.

DoubleToIntFunction

A function that accepts a double-valued argument and produces an int-valued result.

IntBinaryOperator

An operation on two int-valued arguments that produces an int-valued result.

Q& A The type of a Lambda

What are the types for these Lambda expressions?

- Use the previous slide or Java API docs to look up what the different options are

- `s -> System.out.println(s)`
- `(int x, int y) -> { return x + y; }`
- `() -> return "Hello World";`
- `r -> Math.PI * r * r;`
- `(double d) -> (int) d;`

Q& A The type of a Lambda

What are the types for these Lambda expressions?

- Use the Java API docs to look up what the different options are

- a) `s -> System.out.println(s)`
 - `Consumer<String>`
- b) `(int x, int y) -> { return x + y; }`
 - `IntBinaryOperator`
- c) `() -> {return "Hello World";}`
 - () -> "Hello World"
 - `Supplier<String>`
- d) `r -> Math.PI * r * r;`
 - `DoubleFunction<Double>`
- e) `(double d) -> (int) d;`
 - `DoubleToIntFunction`

Qn Using Lambda expressions in Java

Lambda functions can be used in different situations:

- Passed into methods
 - Useful if you need a small method without the overhead of writing the boiler plate
- The forEach() method is available for all the List classes and applies whatever function is passed to the method to each element in the list

```
// using a loop:  
ArrayList<Integer> templist = new ArrayList<Integer>();  
for (int x = 0; x < intlist.size(); x++) {  
    templist.add(intlist.get(x) + 1);  
}  
  
// using lambda:  
intlist.forEach(i -> templist.add(i + 1));
```

QA Streams

Collections have a stream() method available

- This turns the collection into a stream of elements
- Allows a series of operations to be applied to the elements
- It is also possible to create a stream of objects from values using Stream.of(...)
- Streams of numbers can be generated using IntStream.range(1, 4)

Functional methods are available to streams

```
➤ map()
➤ filter()
➤ reduce()
➤ flatMap()
➤ collect()

myList.stream()
    .filter(s -> s.startsWith("c"))
    .map(String::toUpperCase)
    .sorted()
    .forEach(System.out::println);
```

Q map()

The map function takes each element in a stream and applies a function to it

```
List<Integer> intList = List.of(1, 2, 3, 4, 5);  
Stream<Integer> newList = intList.stream().map(i -> i * 2);  
//newList = (2, 4, 6, 8, 10)
```

Maps always return a stream object of the same generic type as the return type of the function

- In this case the function returned an Integer object ($i * 2$)
- The output does not need to match the input

```
Stream<Boolean> isEven =  
    intList.stream().map(i -> i % 2 == 0);  
isEven.forEach(s -> System.out.print(s + ", "));
```

Output:
false,true,false,true,false,

Q & filter()

The filter function creates a new stream based on whether a predicate is true or not

- Take this list and return only the members of the list that are even'

```
Stream<Integer> isEven =  
    intList.stream() .filter(i -> i % 2 == 0);  
isEven.forEach(s -> System.out.print(s + ", "));
```

Output:
2,4,

Map and filter can be combined together to create a new stream of objects

```
intList.stream()  
    .map(i -> i * i)  
    .filter(i -> i % 2 == 0)  
    .forEach(System.out::println);
```

Q4 collect()

The collect() method gathers the output of map() and filter() operations and collects them together into the required form

- By default map and filter return Streams, with collect we can return a list object
- Performs various reduction options on the stream

```
List<Integer> mapFilterList = intList.stream()
    .map(i -> i * i)
    .filter(i -> i % 2 == 0)
    .collect(Collectors.toList());
```

```
String total = intList.stream()
    .map(i -> i * i)
    .filter(i -> i % 2 == 0)
    .collect(Collectors.summarizingInt(i -> i))
    .toString();
System.out.println("Statistics: " + total);
```

Output:

```
Statistics: IntSummaryStatistics{count=2, sum=20, min=4, average=10.0, max=16}
```

Q1 **reduce()**

The reduce method takes a Stream of objects and reduces it down according to some function

- The equivalent of a 'fold' operation in other languages
- Uses an accumulator and applies a function to every element in the list and the current value of the accumulator

```
String concatenated = list.stream()
```

```
.reduce("", String::concat);
```

Output:
abcccd

- The "" is the string it starts with, then every element in the list is concatenated with that string

```
List<Integer> intList = List.of(1,2,3,4,5);  
intList.stream().reduce(1, (x, a) -> a * x);  
//Output: 120
```

Qn Virtual extension methods (default methods)

Virtual extensions allow for default implementations of methods to be included in interfaces

- Uses keyword: **default**
- Override these methods the same way as with standard interfaces
- Allows us to specify particular functionality for a method

```
public interface writer {  
    public default void write(String msg) {  
        System.out.println(msg);  
    }  
}  
  
public interface prettyWriter extends writer {  
    @Override  
    public default void write(String msg) {  
        System.out.println("*****" + msg + "*****");  
    }  
}
```

Qn Using virtual extension methods

Classes can be created using these interfaces

- May override virtual extension methods at declaration time or use the default implementation

```
public class Foo implements writer{  
    public void doStuff(){  
        write("Hello World");  
    }  
}
```

```
public class Bar extends Foo implements prettyWriter {}
```

```
Foo f = new Foo();  
f.doStuff();  
//output: Hello World
```

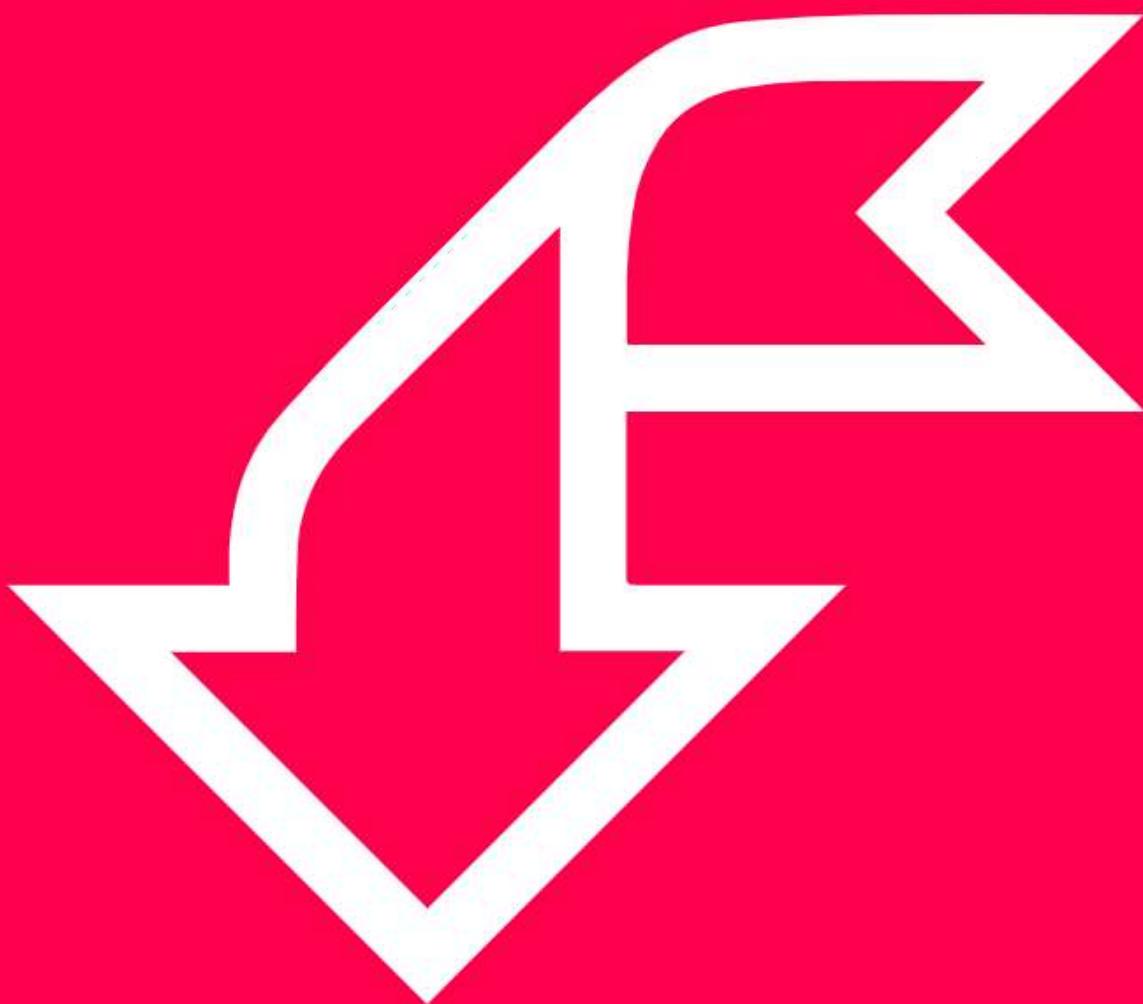
```
Bar b = new Bar();  
b.doStuff();  
//output: *****Hello World+++++
```

Q4 Methods with the same name

If you implement two unrelated interfaces (one does not extend the other) that both have methods with the same name, your code will not compile

- To solve this problem we must override the write() method in class A and tell it specifically what to do
 - Use the super call to access the methods in the UnrelatedInterface interface if required

```
public class A implements prettyWriter, UnrelatedInterface {  
    public void write(String msg) {  
        UnrelatedInterface.super.write(msg);  
    }  
}
```



Exercise

Practice with some of the functional options in Java

- Lambdas and streams
- Finding types for lambdas
- Passing methods as parameters
- Virtual extension methods

SUMMARY

Streams

- map, filter, collect and reduce

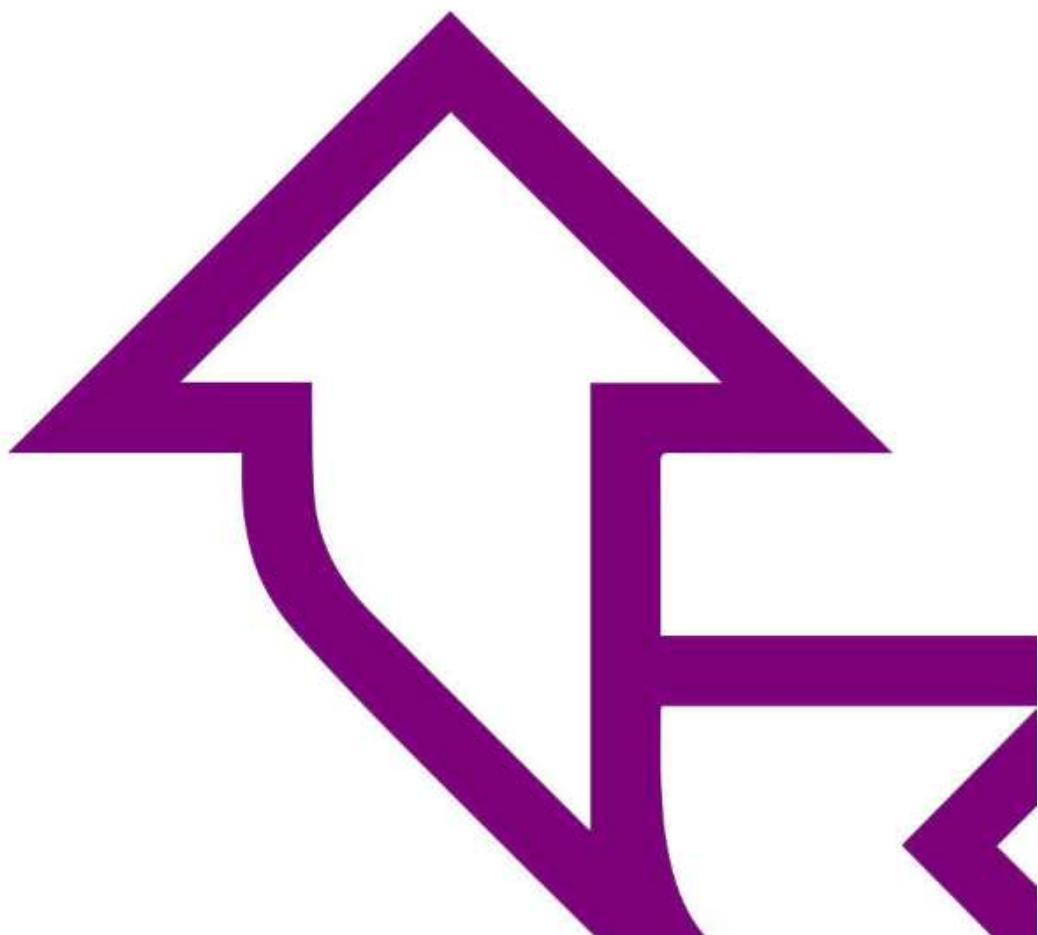
Virtual extension methods

- What and why





Packaging it up



OUTLINE

Distributing software

- Creating Jar files
- Compiling on the command line
- Creating Jars in IntelliJ

Build managers

- Maven
- Other build managers



OBJECTIVES

By the end of this session we should be able to:

- create a jar file from the command line
- create a jar file from IntelliJ
- understand the idea of build managers

Q4 Distributing software

So far we've used IntelliJ to run our programs:

- Now we want to give the source code to users
 - They don't want to open IntelliJ to run their application
- Java can generate JAR files:
- Package the code and libraries into a file
 - Can be executed by clicking on it (like .exe files)
 - Use the command line to launch programs:

```
Packaging>java -jar DG_13.jar
```

Q4 Creating a Jar file

To create a jar file on the command line:

jar cf output.jar input.class

- The options used here are **cf**
 - **c** – create a jar
 - **f** – output to the named file
- Include the **.class** files as input files, not the java files
- The **manifest** tells java how to execute the file
 - It can be automatically generated - better to manually specify it when using the command line
- To view what is inside a jar file use:
jar tf output.jar

```
Packaging>jar tf DG_13.jar
```



```
META-INF/MANIFEST.MF  
com/qa/Car.class  
com/qa/Drone.class  
com/qa/hasPassengers.class  
com/qa/hasPayingPassengers.class  
com/qa/IsDriveable.class  
com/qa/main/  
com/qa/main/VehicleMaker.class  
com/qa/Motorbike.class  
com/qa/Taxi.class  
com/qa/Vehicle.class
```

Q4 Editing the manifest file

To specify the main() method to run in our java application we need to edit the manifest file

- The automatically generated manifest file doesn't include a link to the main method

Create a new text file with a suitable name and include the line:

Main-Class: com.qa.main.VehicleMaker

- This names the class containing the main method to start the program

From the command line run:

```
jar cfm output.jar manifest.txt input.class
```

```
Packaging>jar cfm DG_13.jar manifest.txt com\qa\* com\qa\main\*
```

Then you can run your jar file using the **java -jar** command

```
Packaging>java -jar DG_13.jar
```



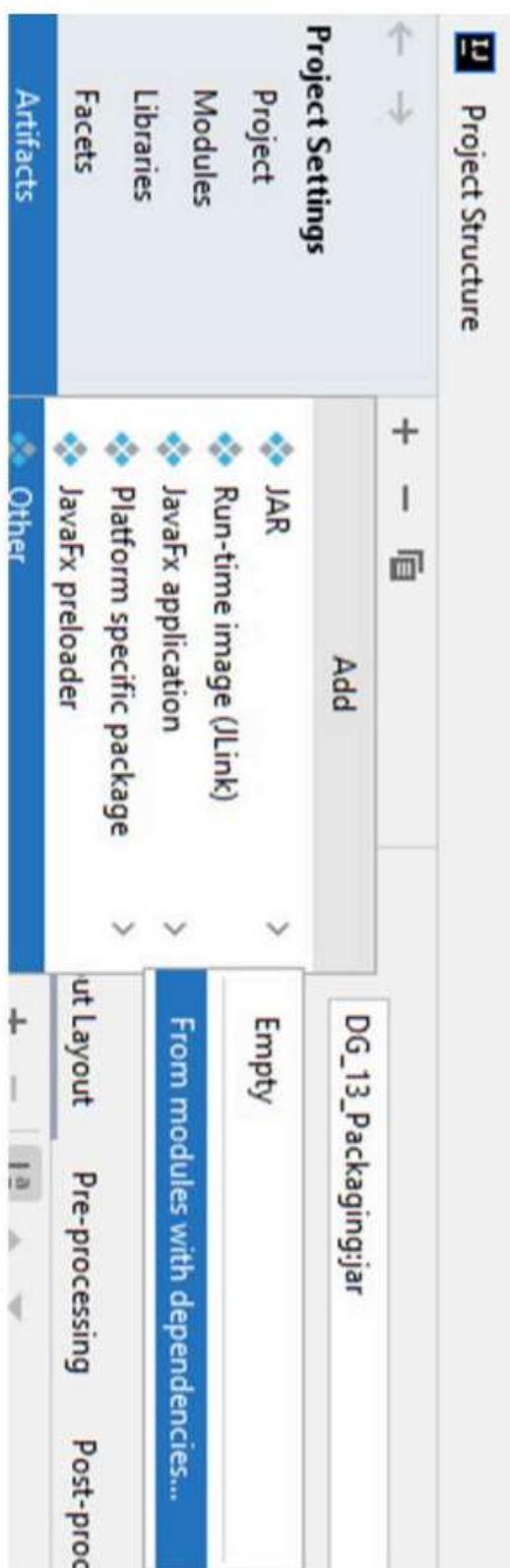
```
Type: class com.qa.Drone; Make: Amazon;
Type: class com.qa.Car; Make: Ford; Model: car1's speed after running cruiseAtSixty;
car1's speed now is: 70
```

Q Creating a Jar from IntelliJ

In intelliJ a built Java archive (JAR) is called an artifact

To create an artifact configuration for a JAR:

- Select File -> Project Structure from the Main menu
- Select Artifacts under Project Settings
- Select Add (+), point to JAR and select From modules with dependencies



Q4 Creating a Jar from IntelliJ

In the Create JAR from Modules dialog

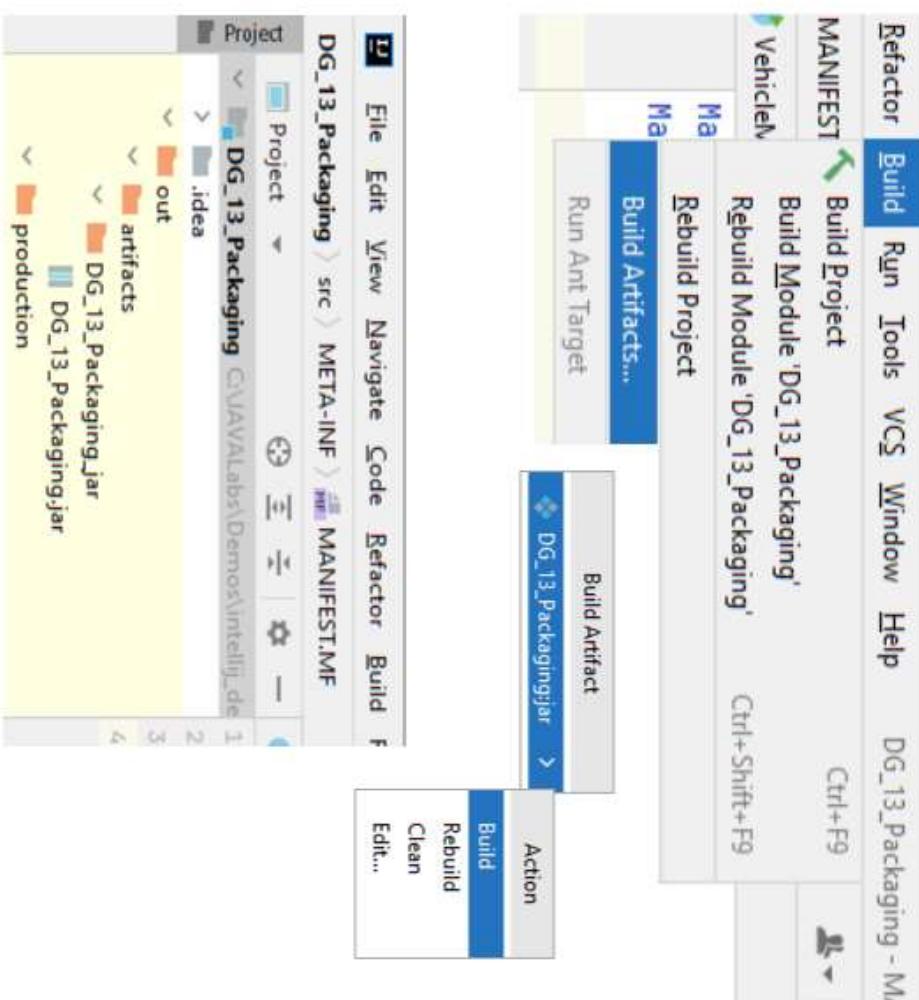
- In the Main Class field, click the Browse button and select the main class for the application
- Ensure the appropriate directory is selected for the manifest
- Select OK
- The artifact configuration is created and shown on the right of the Project Structure dialog



Q4 Building the Jar artifact

From the main menu, select Build -> Build Artifacts

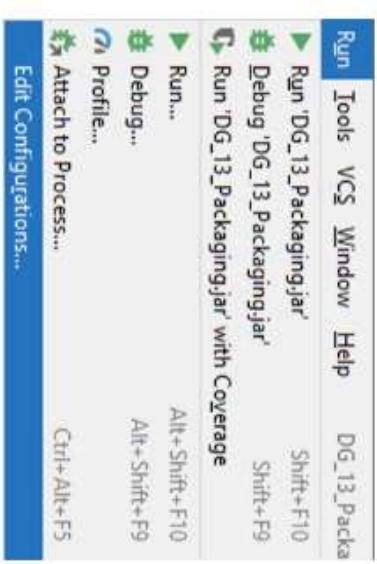
Point to the jar file and select Build



The jar file can now be seen in the Project hierarchy under out/artifacts

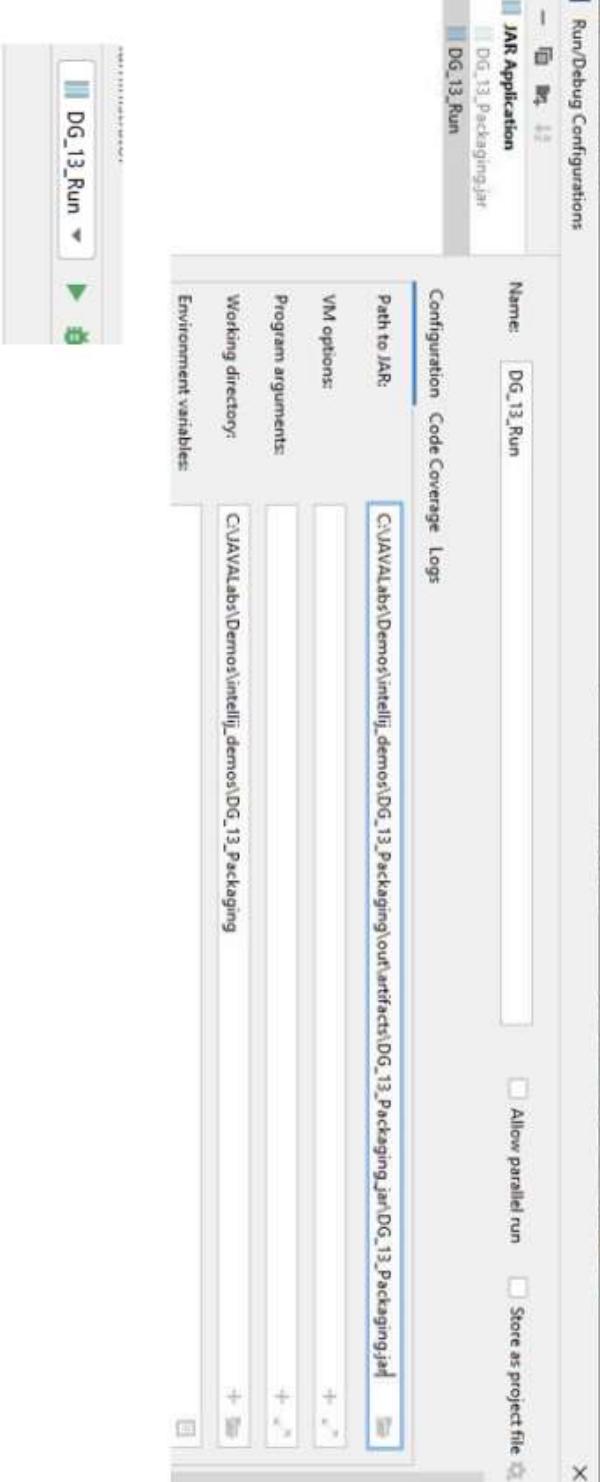
Q4 Running the Jar in IntelliJ

Create a run configuration:



- Select Add and select JAR Application
- Give a name for the configuration
- Select the Path to the JAR

To run, select the configuration and choose Run



Q4 Running the Jar at the Command Line

Navigate to the directory holding the JAR and use the java -jar option

Packaging>java -jar DG_13.jar

```
Type: class com.qa.Drone; Make: Amazon; Model: Deliverer; Colour: Black Fuel: Battery; Altitude: 100
Type: class com.qa.Car; Make: Ford; Model: Mustang; Colour: Red Fuel: Petrol; Passengers : 4
car1's speed after running cruiseAtSixty is: 60
car1's speed now is: 70
car1's speed after running cruiseAtSixty is: 60
Type: class com.qa.Motorbike; Make: Kawasaki; Model: Ninja; Colour: Green Fuel: Petrol
bike1's speed after accelerating is 100
Type: class com.qa.Taxi; Make: Skoda; Model: Octavia; Colour: Silver Fuel: Diesel; Passengers : 5; Fare rate: 2.25
*** What class is taxi1? ***
taxi1 is a Vehicle
taxi1 is a Car
taxi1 is a Taxi
```

BUILD MANAGERS

BUILD MANAGERS

Build managers are a method of handling dependencies in projects

- No need to hunt for the right version of a library
- Compiles to a JAR file as required
- It handles all the dependencies, we just tell it where to find them via the libraries
- Maven is a popular build manager, but there are many available

QA Maven

Apache Maven is:

- 'a software project management and comprehension tool'
- <http://maven.apache.org/>

A dependency manager for projects

- Build automation
- Built on the project object model (POM)

Each project has a pom.xml file

- ❖ Declares how the project should be built
- ❖ What dependencies to include
- ❖ Where to deploy the project to

Can be used at the command line, or through an IDE
Build projects based on archetypes

Q4 Other build management software

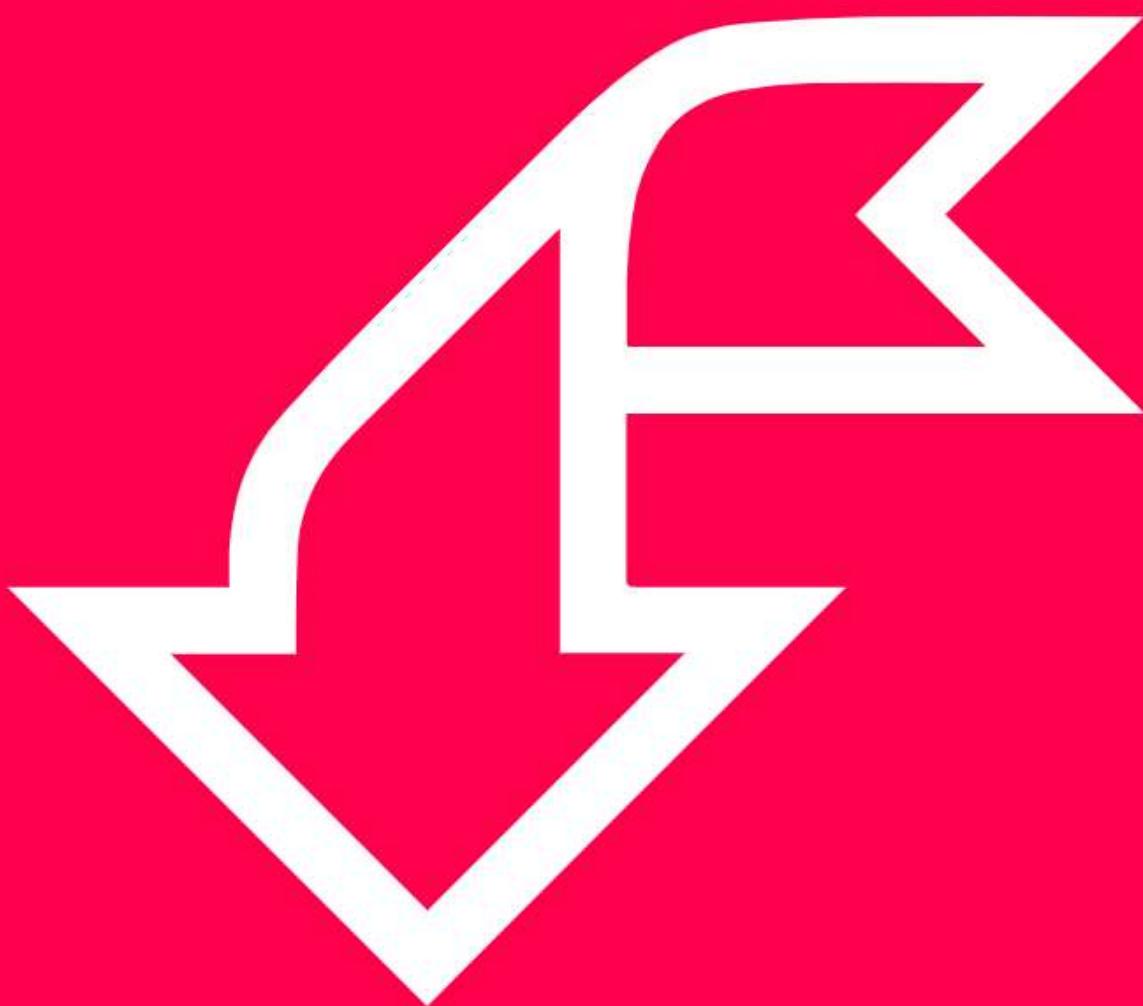
Gradle

- Similar principle to Maven
- No XML
- Groovy based



Jenkins

- Continuous integration
 - Triggers based on time or source changes to recompile, test and upload new projects
 - Can build in source control mechanisms
 - Run the build scripts from other automation methods



Exercise

Generate some Jar files from your projects

- Look at Eclipse and the command line (optional)

SUMMARY

Distributing software

- Creating Jar files
- Compiling on the command line
- Creating Jars in IntelliJ

Build managers

- Maven
- Other build managers



Javadoc



Outline

Javadoc

- What is documentation?
- Why do we want it?
- The Java API docs

Writing our own documentation



OBJECTIVES

By the end of this session we should:

- Know where to go for more information about Java classes
- Be able to write our own documentation
- Generate Javadoc from comments in the code

q

JAVADOC

Q&A What is Javadoc?

Java documentation

- Generated directly from the source code using comments
- Creates a set of HTML documents
- Specific format for comments
- Allows others to understand the classes you have written
- Also allows us to look up how to use classes in the main API:

<https://docs.oracle.com/en/javase/11/docs/api>

The screenshot shows a Java API documentation page for the `java.lang.Double` class. The top navigation bar includes links for OVERVIEW, MODULE, PACKAGE, CLASS, USE, TREE, DOCUMENTATION, INDEX, and HELP. A sidebar on the left lists categories like ALL CLASSES, OVERVIEW, MODULE, PACKAGE, CLASS, USE, TREE, DOCUMENTATION, INDEX, and HELP. The main content area is titled "Module java.base". It shows the `Double` class summary, which includes its inheritance path from `Object` through `Number`, `Comparable`, and `Serializable`. It also lists implemented interfaces: `Double`, `Comparable<Double>`, and `Serializable`. The class is described as supporting the primitive type `double` and providing methods for conversion between `Double` and `double`. The "See Also" section links to `Double`, `Enum<E>`, and `Appendable`.

Field Summary

Q& Why do we want extensive documentation?

- Makes your code easier to understand
- For you now
- For you in the future
- For other people at any point
- Anyone can understand what a method does without looking at its implementation

toString

```
public String toString()
```

Returns a string representation of the object. In general, the `toString` method returns a string that "textually represents" this object. The result should be a concise but informative representation that is easy for a person to read. It is recommended that all subclasses override this method.

The `toString` method for class `Object` returns a string consisting of the name of the class of which the object is an instance, the at-sign character '@', and the unsigned hexadecimal representation of the hash code of the object. In other words, this method returns a string equal to the value of:

```
getClass().getName() + '@' + Integer.toHexString(hashCode())
```

Returns:
a string representation of the object.

WRITING OUR OWN DOCUMENTATION

QA Writing our own Javadoc

Javadoc is generated from the comments in the code preceding a class, field or method

- The second star at the start of the comment identifies it to the Javadoc compiler

```
/**  
 * This is a javadoc comment  
 */
```

A doc comment is made up of a description followed by block tags

- These tags are used to generate formatted comments in the eventual webpage created

@param	@author	@since	@serial
@return	@version	@throws	@serialField
@see	@deprecated	@exception	@serialData
{@link}			

QA Writing our own Javadoc

IntelliJ will attempt to help you write the doc comment as much as it can

- If you start a comment with `/**` it will suggest any tags it considers applicable when you type `@`

`@author`

- Is added to doc comments at the class level to identify the author of the class



Q4 Writing Javadoc for methods

Generally you want to generate Javadoc for public methods

- As private methods are not accessed outside the class it is usually unnecessary to document them
- Parameters for the method - include the name and a short description

```
/*
 * Constructor for primary type of vehicle in the company's fleet
 * All parameters currently unrestricted - restrictions planned Q3 2023
 * Fuel types petrol and Diesel may have limitations
 *
 * Parameters:
 * @param make - Manufacturer of the car (eg. Toyota, Renault, BMW)
 * @param model - Car model name (eg. Prius, Megane, 2 Series Gran Coupe)
 * @param colour - The major colour of the vehicle (eg. Borley Gold, Cappuccino, Red)
 * @param fuelType - Source of power (eg. Battery, Petrol, Electric, Diesel, Hydrogen, Hybrid)
 * @param wheels - Number of wheels on the vehicle - generally 4, but 3 (eg. Nimbus) and 6 (Mercedes G63 AMG)
 */
2 usages
public Car(String make, String model, String colour, String fuelType, int wheels) {
```

Q4 Writing Javadoc for methods

@return

- What is returned from a method, including the type

```
/**  
 * Returns the title of the book  
 * @return String title  
 */  
public String getName() {  
    return name;  
}
```

Q A Javadoc for methods

In the description of a method you can also explain what will happen if errors occur

```
/**  
 * Add an individual author to the list of authors of the book  
 * Will print an error if there are already 5 listed authors  
 * @param author the author to add to the array  
 */  
public void addAuthor(String author) {  
    if (authorNum < 5) {  
        authors[authorNum] = author;  
        authorNum++;  
    } else {  
        System.out.println("Error, you can only have up to five  
        authors. Change not made");  
    }  
}
```

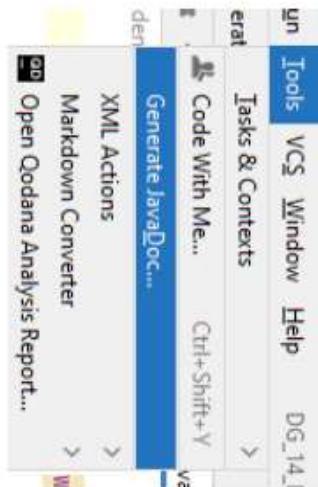
QA Tags

Annotation	Description	Example
@param	Parameters for a method	@param name title of book
@return	Return from a method	@return String title of book
@see	See another class for additional information	@see class
@author	Author of the class, added at class level	@Author Admin
@version	Version of this class	@Version %I%, %G%
@deprecated	If the method has been marked for removal	@deprecated
@since	When the class was added	@since 1.0
@throws	If the method throws an exception (same as @exception)	@throws Exception
@serial	Documents serialised fields and data	@serial
@serialField		
@serialData		
{@link}	A link to another javadoc place (inline, vs in its own section)	{@link package.class#member label}

QA Generating the Javadoc

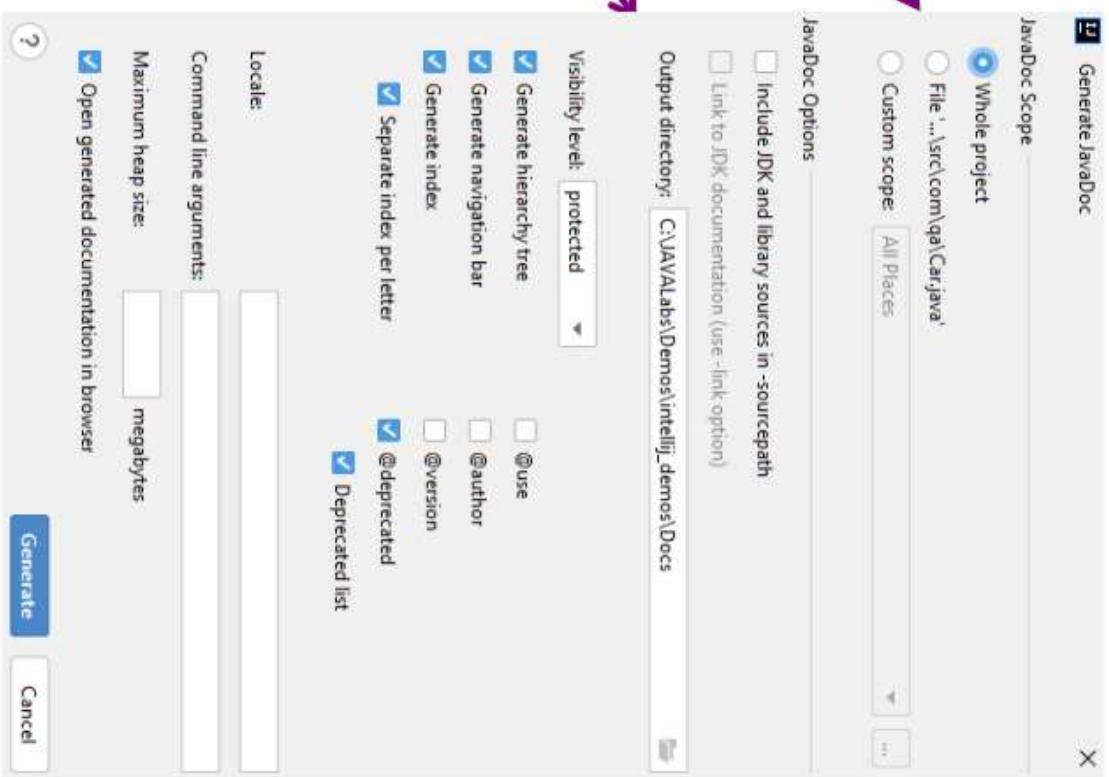
IntelliJ can generate the Javadoc for us

- Tools menu | Generate Javadoc



Scope to use

Where to save
the Javadoc



QA Generating the Javadoc

On generation of the Javadoc

- The console will tell you of any errors in your Javadoc comments
- The HTML files will be generated for you regardless based on the doc comments you provided
- IntelliJ will automatically open the docs in the browser

To access the docs subsequently

- In File Explorer locate the directory where the Javadoc was saved
- Click index.html
- The docs will open in the Web browser



QA The generated docs

Use the Search field or select on the links to navigate through the docs

The screenshot shows a Java documentation interface with several panels and navigation arrows.

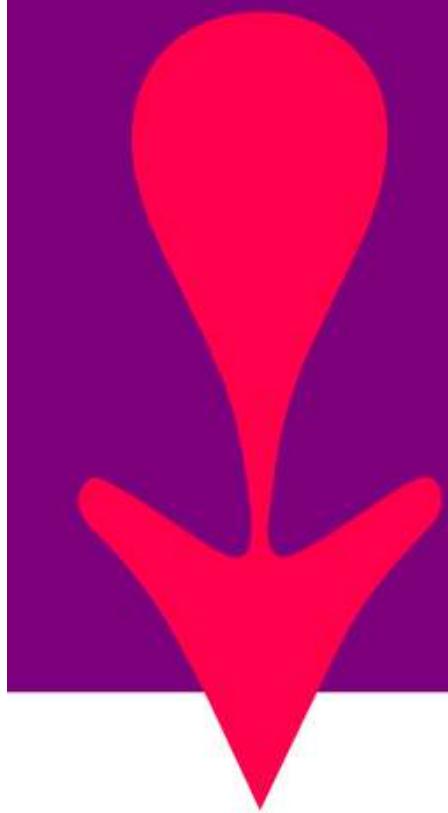
- Top Bar:** OVERVIEW, PACKAGE (highlighted in orange), CLASS, TREE, INDEX, HELP.
- Left Sidebar:** PACKAGE DESCRIPTION | RELATED PACKAGES | CLASSES AND INTERFACES.
- Search Bar:** SEARCH:
- Package com.qa.main:** package com.qa.main
Related Packages: Package com.qa Description
- Class VehicleMaker:** Class VehicleMaker
java.lang.Object
com.qa.main.VehicleMaker
extends Object
- Description:** Main class for managing vehicle fleet Primary vehicle classes of the business - Car, Motorbike Additional vehicles managed - Taxi, Drone Updated 18/11/2022: JavaDoc comments added Author: Admin updated 18/11/2022: JavaDoc comments added
- Constructor Summary:** Constructor Summary
- Method Details:** Method Details
- VehicleMaker:** Main class for managing vehicle fleet Primary vehicle classes of the business - Car, Motorbike Additional vehicles managed - Taxi, Drone Updated 18/11/2022: JavaDoc comments added
- Constructor:** Constructor Description
- VehicleMaker():** VehicleMaker()
- Method Summary:** Method Summary
- All Methods:** All Methods (highlighted in orange) Static Methods Concrete Methods
Modifier and Type Method Description
- main:** public static void main(String[] args)
Main activities of managing the organisation's fleet of vehicles
Parameters:
args -- not currently in use
- static void main(String[] args):** Main activities of managing the organisation's fleet of vehicles

Exercise

Write Javadoc comments for your book class or other classes created in earlier exercises

Generate the Javadoc for these and have a look

Examine some of the classes in the Java API

A large, stylized graphic of a flower or heart shape, composed of overlapping red and blue curved and triangular shapes, centered at the top of the slide.

QA

Summary

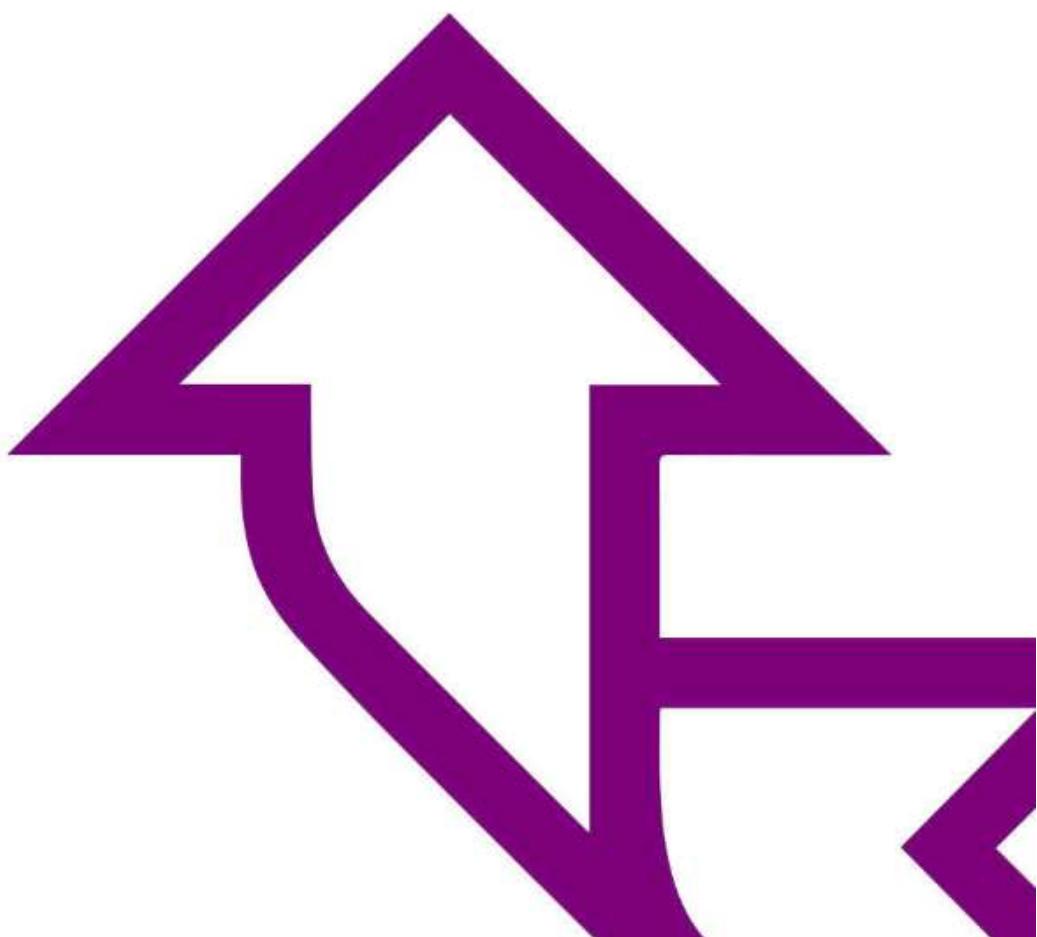
Javadoc

- What is documentation
- Why do we want it
- The Java API

Writing our own documentation



Test-driven Development (TDD)



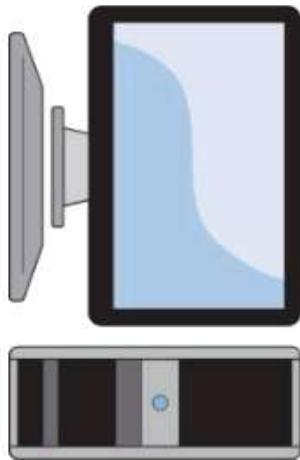
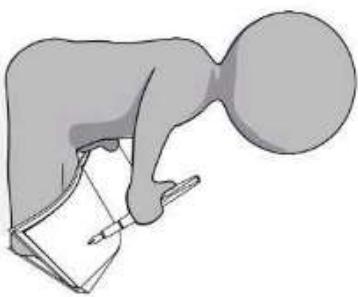
OUTLINE

Test-driven development with Junit

- Automated testing or manual testing
- Unit testing
- Test-driven development
 - TDD life cycle – red, green, refactor
 - JUnit annotations
 - Assert methods and test class
- Creating and executing the test cases



QA Automated or manual testing



Manual testing

- Difficult to repeat tests in a consistent manner
- Can't guarantee that in regression testing same values are re-entered
- In speed tests, it's difficult for an operator to match a computer
- Can only be executed by certain people

Automated testing

- Can be executed by anyone
- Perfect for regression testing
- Series of contiguous testing can be done, where the results of one test rely on another
- The build test cycle is increased

QA Manual tests

Write a test harness for the class under test

- **Main method creates instance of class, invokes method, System.out.println()**

Drawbacks

- **Manual inspection to see if code performed correctly**
- Error prone
- Not scalable
- **Do not aggregate (x out of y tests passed)**
- **Do not indicate how much of the code was exercised**
- **Do not integrate with other tools – build process, CI**
- “main()” does not tell you what the scenario is
- **Not extensible**

QA Unit testing

A unit can be

- A method
- A database query, stored proc or transaction
- A dynamic web page

A unit test

- Tests one behaviour of an object
- Is automated, self-validating, consistent/repeatable, independent, readable, easy to maintain and fast

The JUnit Framework

- Common design
- Setup, test, assertion; suites of tests

QA Test-driven development

Test written **before** implementation

- Tools and techniques make TDD a very rigorous process

Developers become Test infected – cannot program without test first

Test – Code – Refactor

- Write new code only if you first have a failing automated test
- Eliminate duplication

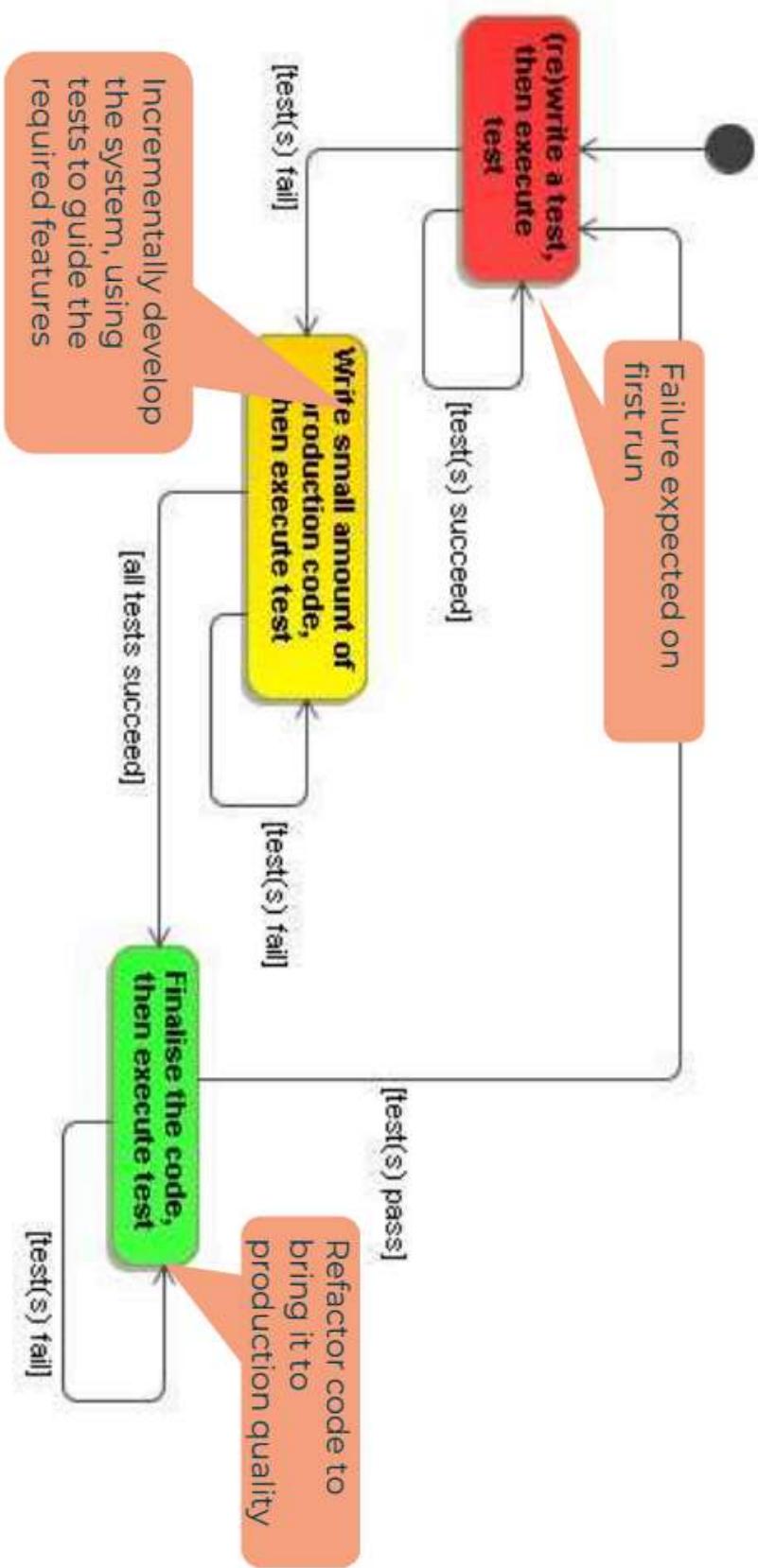
Uses a **Red – Green – Refactor** workflow

Red – test failure

Green – success

Refactor – change a little of the code, but not the functionality

QA TDD Life Cycle – Red-green-refactor workflow



TDD process

First write the test

- Design the API for the code to be implemented
- Using an API in tests is best way to evaluate its design

Write just enough code for test to pass

- Minimises code bloat
- Keeps developer focussed on satisfying the requirement embodied in the test

Refactor

- change some code without changing functionality
- Develop in small iterations



QA TDD worked example

Find the highest number in an array of ints

1. Start by writing a test (e.g. `ArrayUtilsTest` class)

- What's a good simple starting case?

Find the highest in an array with one number

- What should we call the test method?

```
findHighestInArrayOfOne()
```

- How do we express this test – what are we asserting?

```
assertEquals(10, Arrayutils.findHighest(array));
```

- Arrange any fixtures

```
int[] array = {10};
```

QA TDD worked example

2. Make it compile (do just enough)

- public static int findHighest (int [] numbers) { }
- Generally, return 0 or null, etc.

3. Make it fail (run test and verify red Tests failed)

- Using matchers: Expected: is <10> got: <0>

4. Make it pass (do minimum to go green – green ticks)

- return 10;

5. Make it right (remove duplication)

- Remove dependency of code on test
- Duplication test data in solution code
- return numbers [0];

QA TDD worked example

6. Devise next test: what's next bit of functionality?

- findHighestInArrayOfTwo()
- Make it fail (i.e. don't make array {20, 10})

7. Solution: need to handle variable length array

```
int highestSoFar = Integer.MIN_VALUE;  
for (int i = 0; i < numbers.length; i++) {  
    if (numbers[i] > highestSoFar)  
        highestSoFar = numbers[i];
```

8. Triangulate – add further tests

- E.g. findHighestInArbitraryArray()

9. Ensure corner cases are covered. What about {} ?

TDD Benefits

Build up a library of small tests that protect against regression bugs

Extensive code coverage

- No code without a test
- No code that is not required

Almost completely eliminates debugging

- More than offsets time spent developing tests

Tests act as developer documentation

Confidence not fear

- Provides confidence in quality of the code; confidence to refactor



Benefits of refactoring

- Makes code easier to understand
- Improves code maintainability
- Increases quality and robustness
- Makes code more reusable
- Typically to make code conform to a design pattern
- Many now automated through IntelliJ, Eclipse, etc.



Refactoring ≠ Rewriting

vi

JUNIT

QA JUnit: Java xUnit framework

- JUnit Jupiter is the API for writing tests using JUnit5
 - org.junit.jupiter.api
 - Has a set of annotations for configuring tests and extending the framework, for example:
 - @Test
 - Denotes that a method is a test method
 - @ParameterizedTest Denotes that a method is a parameterized test method
 - @Timeout
 - Fail a test if the execution exceeds the specified duration
 - org.junit.jupiter.api.Assertions
 - contains assert() methods including assertThrows() and assertAll()
 - overloaded methods support error messages to be printed if the test fails
 - Junit 5 can be run from IDEs (IntelliJ, Eclipse, NetBeans) and has a Console Launcher to launch from the command line

QA JUnit assertions

A collection of utility methods that support asserting conditions in tests

- Methods are overloaded – for example:

```
assertEquals(boolean expected, boolean actual)
```

```
assertEquals(Object expected, Object actual)
```

```
assertEquals(int expected, int actual, String message)
```

- Using the String version, on failure, the exception is thrown with message

- Remember order: expected then actual

- Paired methods

```
assertSame / assertNotSame(Object expected, Object actual, String message)
```

```
- asserts that the actual and expected reference is/ is not the same
```

```
assertTrue / assertFalse(boolean condition, String message )
```

```
- asserts that condition is true/ false
```

```
assertNull / assertNotNull(Object actual, String message)
```

```
- asserts that actual is/is not null
```

A method can be called to throw an AssertionFailedError on reaching an undesirable point in the test method

```
fail(String message)
```

Q4 JUnit test class and life cycle methods

A class is deemed a test class if it contains at least one method having a JUnit annotation

A test class name will typically be the name of the class to be tested followed by Test

```
class CalculatorTest {...}
```

It may have 1 or more @Test methods (names may optionally begin with test)

The method body should invoke the code being tested and include an assert() method

```
@Test  
void add() {  
    assertEquals(4, Calculator.add(2, 2));  
}
```

It may also have methods to initialise state prior to tests and carry out actions after tests

```
@BeforeAll  
void setUp() {...}  
@AfterAll  
void tearDown() {...}
```

QA Example – class under test

```
public class Person implements Comparable<Person> {  
    private String givenName;  
    private String familyName;  
    private int age;  
  
    // 3 arg constructor, getters and setters, etc.  
    @Override  
    public int compareTo(Person other) {  
        return this.age - other.age;  
    }  
    @Override  
    public String toString() {  
        return familyName + ", " + givenName + " [" + age + "]";  
    }  
}
```

QA Example – define your tests

```
public class PersonTest3 {  
    Person fred;  
    Person bill;  
    Person jane;  
  
    @BeforeEach  
    void setUp() {  
        fred = new Person("Fred", "Foggs", 29);  
        bill = new Person("Bill", "Boggs", 31);  
        jane = new Person("Jane", "Joggs", 29);  
    }  
  
    @Test  
    public void testCompareTo() {  
        assertTrue(fred.compareTo(bill) < 0, "fred is 'before' bill");  
        assertTrue(bill.compareTo(jane) > 0, "bill is 'after' jane");  
        assertEquals(0, fred.compareTo(jane), "fred and jane are equivalent");  
    }  
}
```

There should only
be one assertion in
a test method

JUNIT TESTING IN INTELLIJ

Q4 Create a Maven project to run JUnit

The screenshot shows the IntelliJ IDEA interface with two windows open:

- New Project Dialog:** A modal window titled "New Project" is displayed. It has the following settings:
 - Name:** JUnit_Proj
 - Location:** C:\JAVA\labs\labs
 - Language:** Java
 - Build system:** Maven
 - JDK:** 17 Oracle OpenJDK version 17
- pom.xml (JUnit_Proj) Editor:** The XML editor shows the generated pom.xml file. A context menu is open over the XML code, with the "Add" option highlighted.

Annotations and arrows provide instructions:

- An arrow labeled "Select Add dependency" points to the "Add" option in the context menu.
- An arrow labeled "Enter" points to the search bar in the "Dependencies" dialog.
- An arrow labeled "Click" points to the "Add" button in the "Dependencies" dialog.

Dependencies Dialog: A modal window titled "Dependencies: Manage" is shown. It lists "All Modules" and "JUnit_Proj". In the search bar, "org.junit.jupiter:junit-jupiter" is typed. The results show:

- JUnit Jupiter (Aggregated) org.junit.jupiter:junit-jupiter:5.9.0
- JUnit Jupiter Engine org.junit.jupiter:junit-jupiter-engine:5.9.0
- JUnit Jupiter API org.junit.jupiter:junit-jupiter-api:5.9.0

A "Add" button is visible at the bottom of the list.

pom.xml Content: The pom.xml file contains the following code, with the JUnit dependency highlighted:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <groupId>org.junit.jupiter:junit-jupiter</groupId>
  <artifactId>junit-jupiter</artifactId>
  <version>5.9.0</version>
</project>
```

Q4 Create a Junit test class

The screenshot shows a Java code editor with the following code:

```
public class calculator {
    static double add(double a, double b) {
        return a + b;
    }
}

class calculatorTest {
    @BeforeEach
    void setup() {
    }

    @Test
    void add() {
    }
}
```

A context menu is open over the word "calculator". The menu items are:

- Show Context Actions
- Paste
- Copy / Paste Special
- Column Selector
- Ctrl+V
- >
- Public class Calculator {
- >
- > Create Subclass
- > Create Test (highlighted)
- > Make 'Calculator' package-private
- > Seal class
- >

Below the code editor, a "Create Test" dialog is displayed:

- Testing library: JUnit5
- Class name: CalculatorTest1
- Superclass: (empty)
- Destination package: (empty)
- Generate:
 - setUp/@Before
 - tearDown/@After
- Generate test methods for:
 - add(operands:double...);double
 - multiply(operands:double...);double
- Show inherited methods:
- OK
- Cancel

The "Create Test" button is highlighted.

Below the dialog, the text "Edit generated code" is visible.

QA Running a test

The screenshot shows a Java code editor and a test runner interface.

Code Editor:

```
0  class CalculatorTest2 {  
1      @Test  
2      @DisplayName("Add 3 numbers")  
3      void add() {  
4          assertEquals(expected: 6, Calculator.add(1, 2, 3));  
5      }  
6  }  
7    
8  class CalculatorTest3 {  
9      @Test  
10     @DisplayName("Multiply 3 numbers")  
11     void multiply() {  
12         assertEquals(expected: 8, Calculator.multiply(...operands: 2, 2, 2)),  
13         () -> assertEquals(expected: -8, Calculator.multiply(...operands: 2, -2, 2)),  
14         () -> assertEquals(expected: -8, Calculator.multiply(...operands: -2, -2, -2)),  
15         () -> assertEquals(expected: 6, Calculator.multiply(...operands: 3, 2, 1)),  
16         () -> assertEquals(expected: -6, Calculator.multiply(...operands: 3, 2, -1));  
17     }  
18 }
```

Test Runner:

Run: CalculatorTest3

Test	Status	Message
CalculatorTest3	✓	Passed: 1 of 2 tests - 29 ms
Add 5 numbers	✓	Passed: 1 of 2 tests - 22 ms
Multiply 5 numbers	✗	Failed: 1 of 2 tests - 7 ms

"C:\Program Files\Java\jdk-17\bin\java.exe"

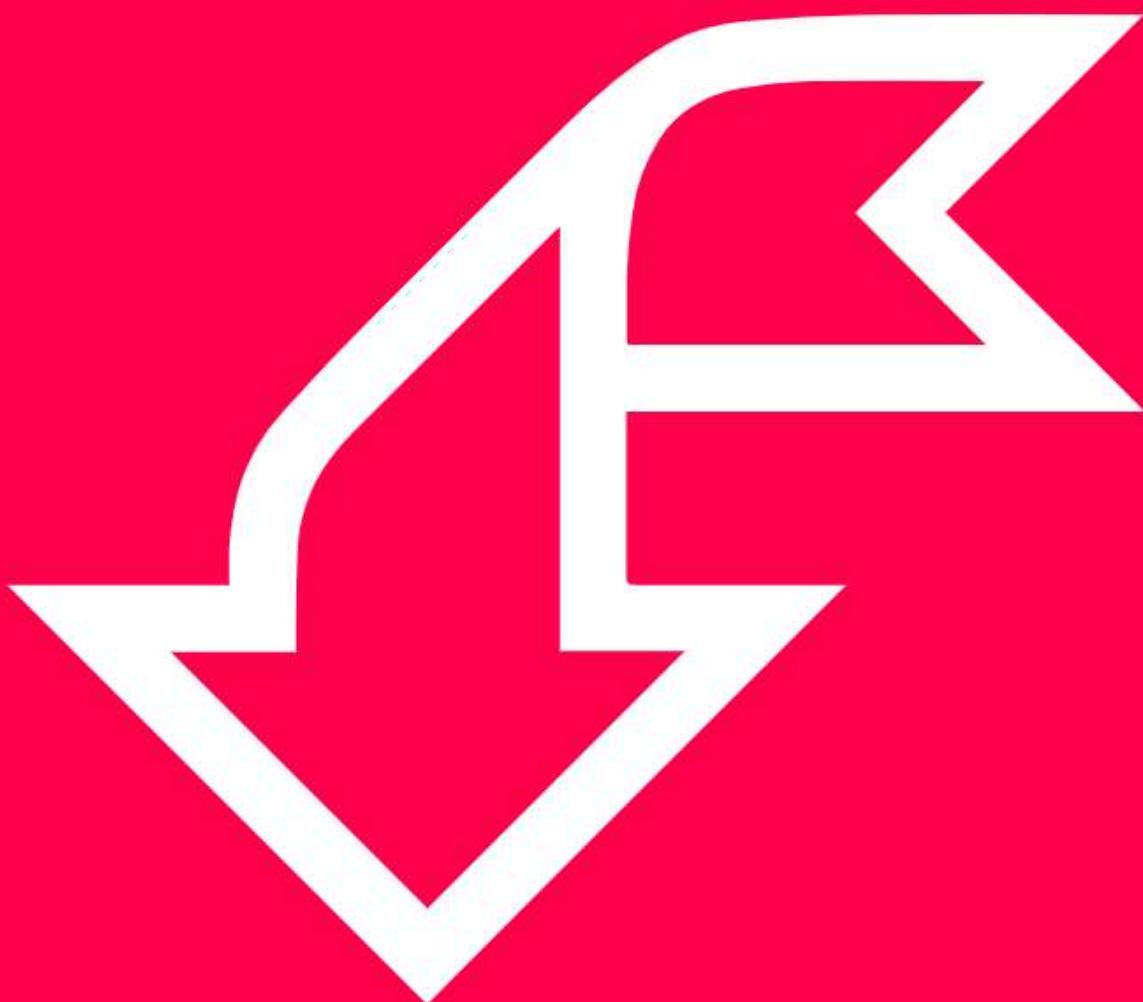
expected: <-230.0> but was: <-240.0>
Comparison Failure:
Expected : -230.0
Actual : -240.0
[Click to see difference](#)

Select and Run all tests or a specific test

SUMMARY

Test-driven development with Junit

- Automated testing or manual testing
- Unit testing
- Test-driven development
 - TDD life cycle – red, green, refactor
 - JUnit annotations
 - Assert methods and test class
- Creating and executing the test cases



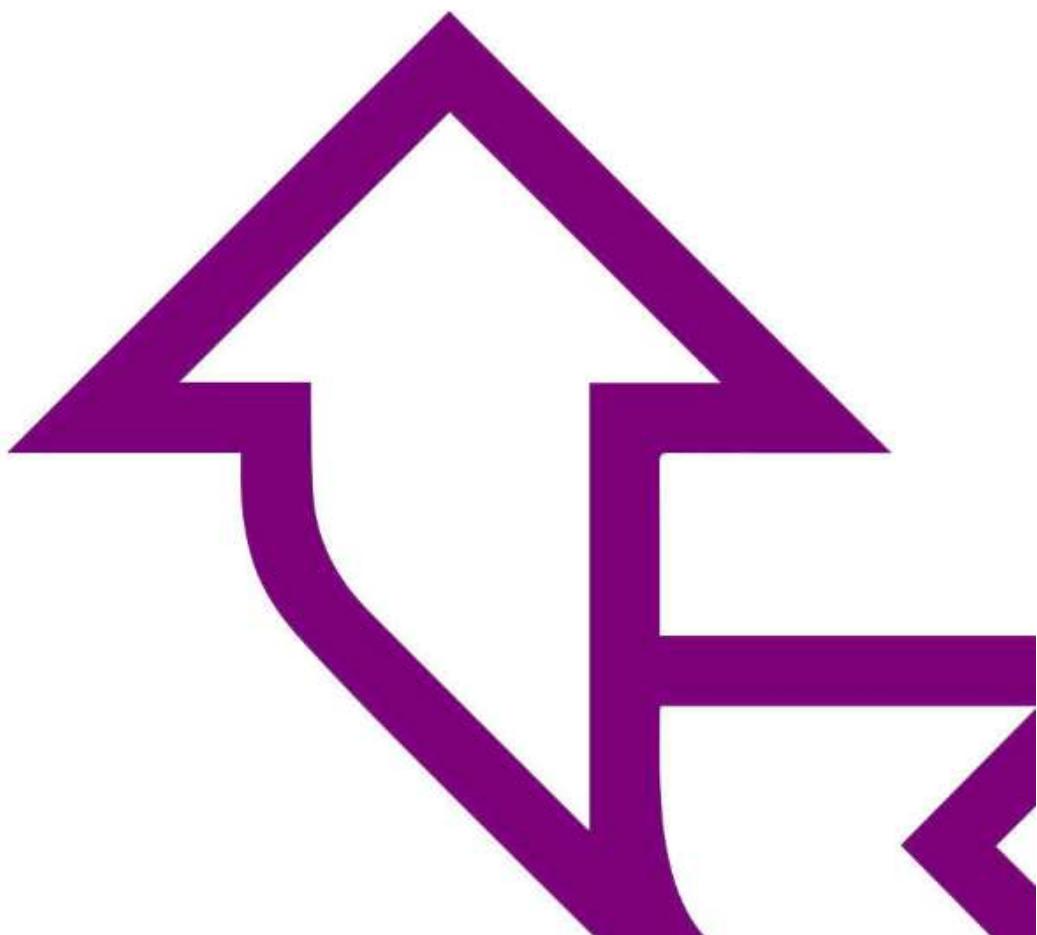
Exercise

Use JUnit in IntelliJ to create a test for
the `compareTo` method of an
Account

- Devise further tests for other Account methods



Introduction to Java modules



QA

OUTLINE

Overview of modules in Java

Module dependencies

Modular JDK



QA

OBJECTIVES

By the end of this session, we should be able to

- Understand the use of Java modules
- Define module dependencies

JAVA MODULES – AGGREGATE PACKAGES

Before Java 9:

Classes could only be organised in packages

- Built into JAR files for distribution
- Public classes are available to all other java code
- No way to organise the packages

From JDK 9:

We can design packages in modules

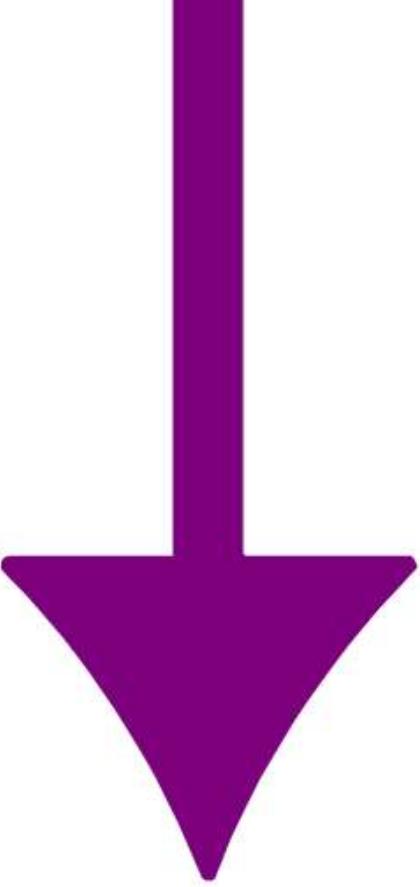
JAVA MODULES – FROM JDK 9

A module is a set of packages

- Designed for reuse
- Provides a higher level of aggregation above packages
- Can include other related files
- Is a program unit above packages and classes
- Can control the visibility of packages
 - to provide strong encapsulation
 - made visible to other modules – exported
 - Internal to this module – concealed

The module system is usable at all levels

- The JDK, libraries, and applications
- Help make applications more reliable and secure
- Can create runtime images with minimal JDK elements



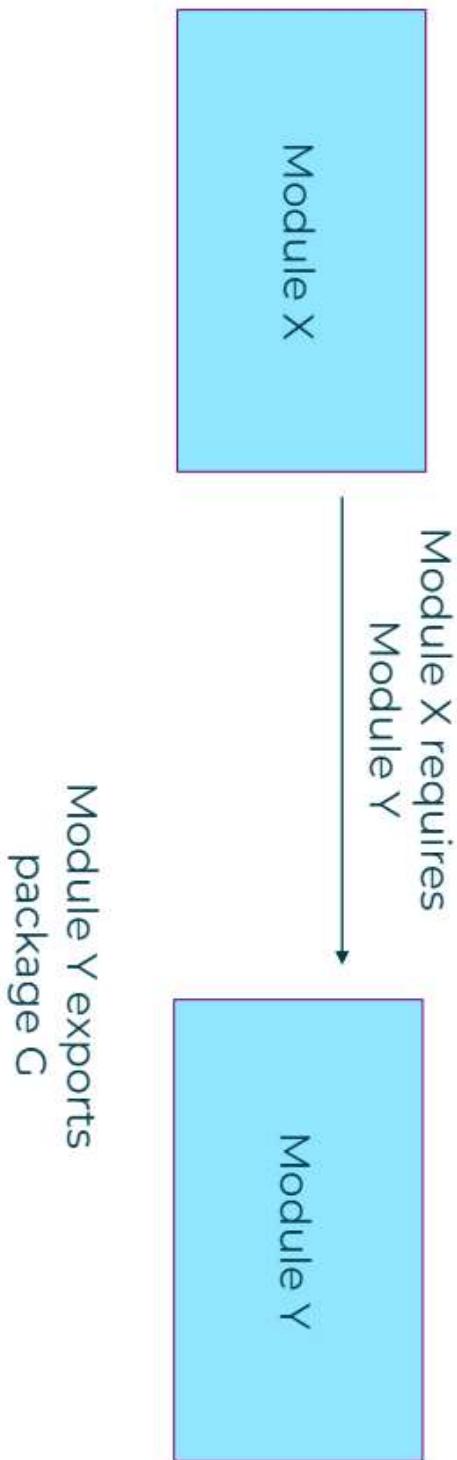
A module is the unit of distribution

- Can be distributed as a JAR file
 - Defines its dependencies on other modules
 - Keeps **all packages concealed** within the module
unless explicitly exported
 - Is a full Java component
 - The **public** modifier only means available to everyone in an exported package
- ## Java module advantages

Q4 Access between modules

Interaction between modules is achieved by

- The 'calling' module, X, specifying that it requires resources from the 'providing' module, Y.
- The 'providing' module, Y, exporting the packages it wants to make visible
- Methods in classes in Module X now have access to public classes in package G of Module Y



MODULE DEPENDENCIES

Q4 Module dependencies

Each module has a module-info.java file

- In the base directory of the module
- It compiles to a module-info.class – the module descriptor file
 - This is where the requires and exports module directives are coded.
 - Could be empty if the module is a self-contained application

```
module modulename{
```

```
}
```

```
module X{  
    requires Y;
```

```
}
```

```
module Y{  
    exports G to X;
```

```
}
```

qa

MODULAR JDK

Q4 Modular JDK

The JDK has been modular since Java 9

- Consists of over 90 modules
- More scalable to small devices
- More secure and maintainable
- Better application performance
- JSE standard modules include
 - java.base
 - java.sql
 - java.xml
- Each module has well defined functionality

The `java.base` module

The most fundamental module

- Every other module depends on it
- It is implicitly required by every module
- `java.base` requires no other module
- It exports all core packages, including `java.lang`,
`java.io` and `java.util`

A large, stylized red flower logo is positioned in the top left corner of the slide. It has a dark red center with several petals, some of which are solid red and others are outlined in red.

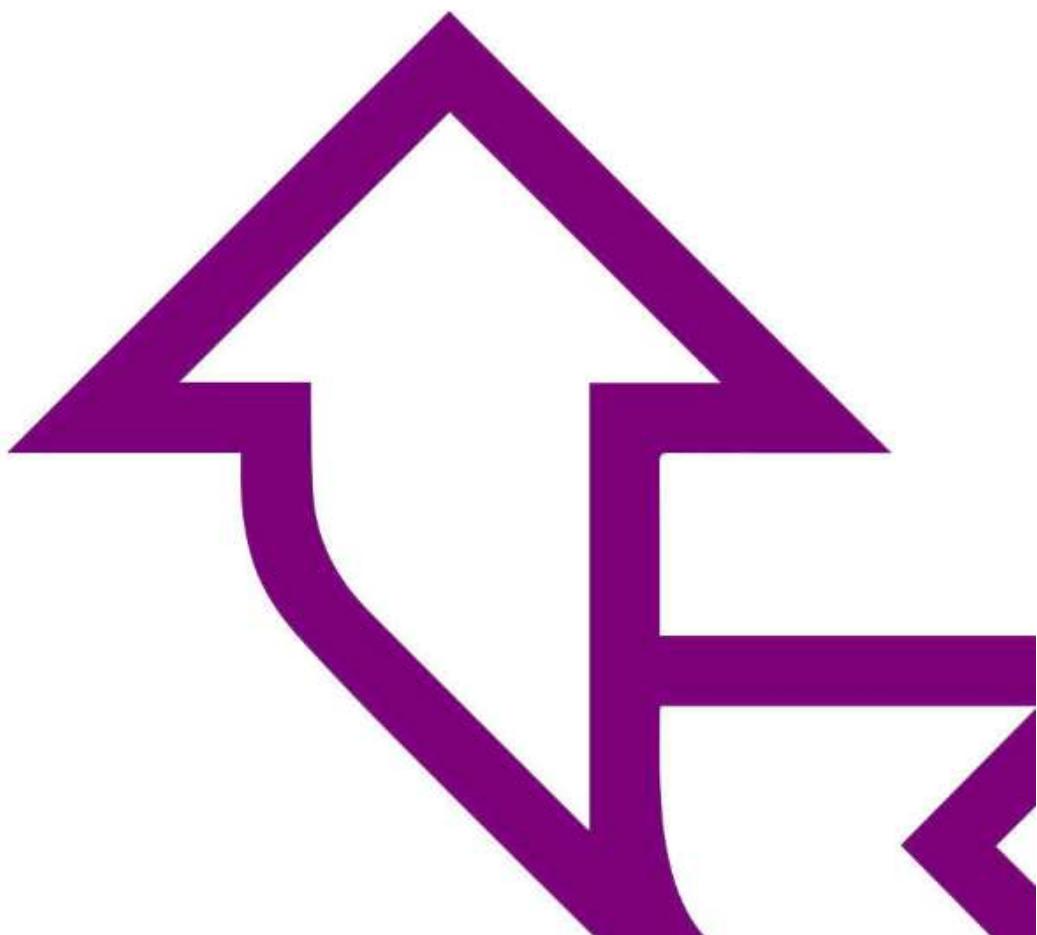
QA

SUMMARY

- Overview of modules in Java
- Module dependencies
- Modular JDK



Introduction to Maven





Installing Maven

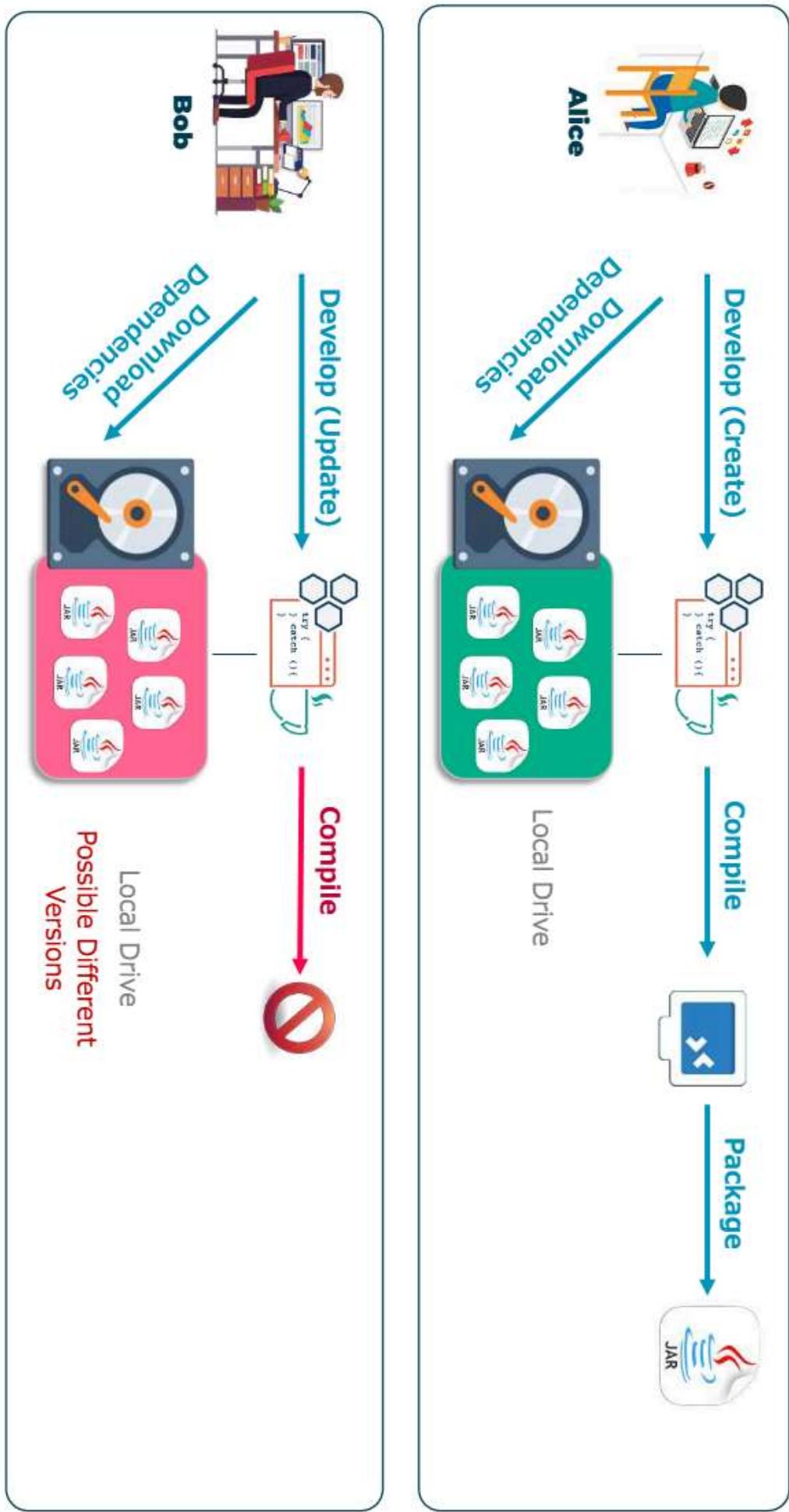


<https://maven.apache.org/install.html>

1. Download the latest version of Maven from the Apache Maven website (<https://maven.apache.org/download.cgi>).
2. Extract the downloaded zip file to the directory where you want to install Maven. For example on Windows, you can extract it to "C:\Program Files\apache-maven".
3. Add the Maven installation directory to the PATH variable.
4. Open a new command prompt and type "mvn -v" to verify that Maven is installed and working properly. You should see the Maven version and a list of available commands.

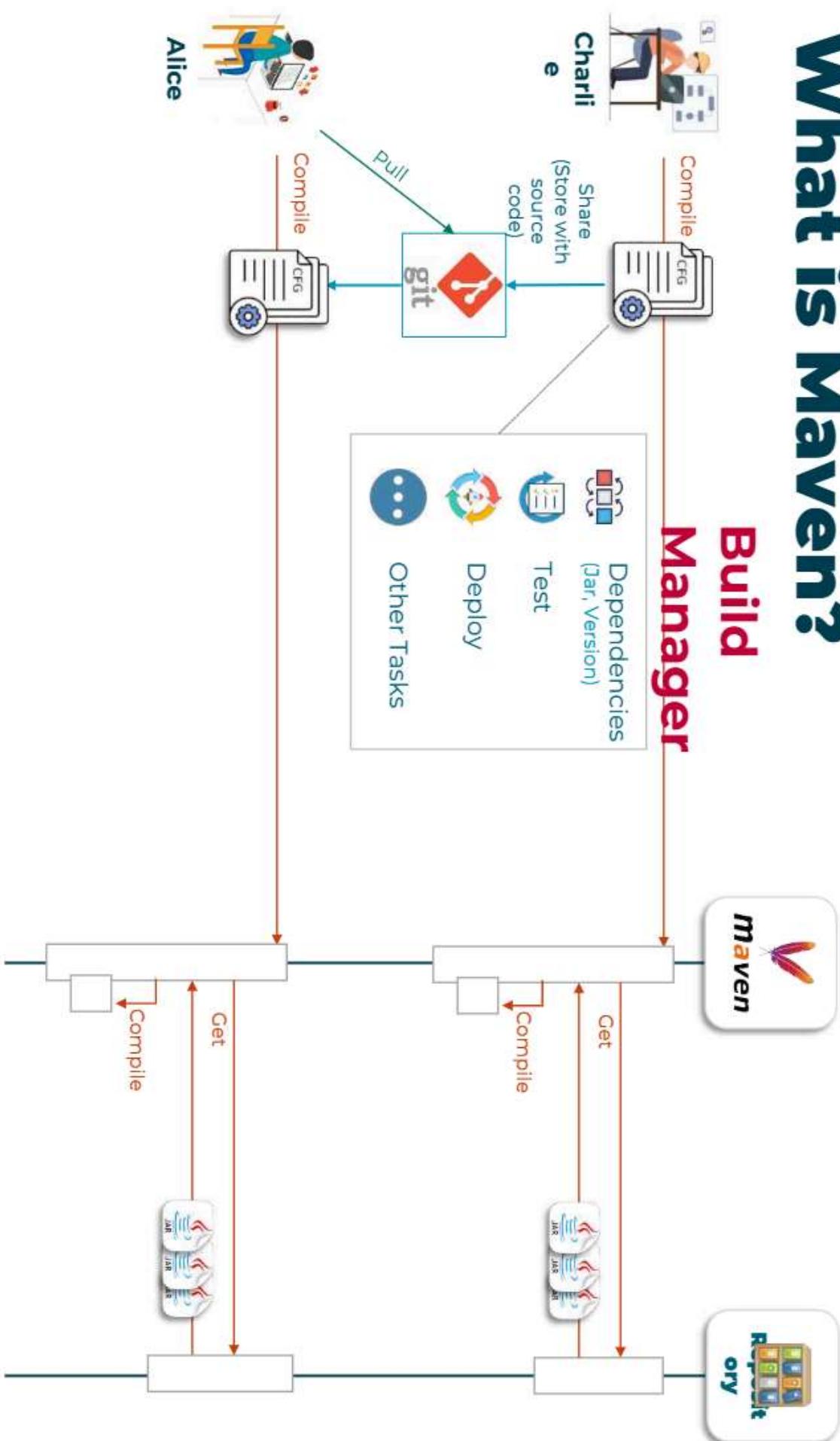
Note: Maven is a command line tool that is invoked from the command prompt or shell

QA Manual software assembly and packaging



Q A What is Maven?

Build Manager



QA About repositories



Repository

In Maven, a repository is a directory where all the **project jars, library jars, and plugins are stored**. Maven repositories can be either local or remote.



Remote Repository

A **remote** repository is a repository that is stored on a remote server. Remote repositories are typically accessed over the internet, and they are used to store artifacts that are shared by multiple projects. For example, **Maven Central** repository is a remote repository that contains a large number of commonly used libraries



Local Repository



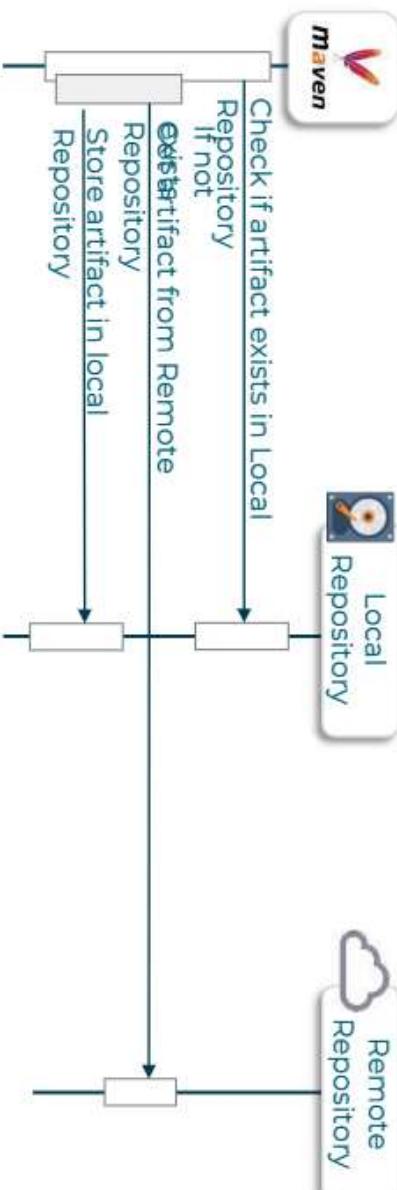
Local Repository

A **local** repository is a directory on the developer's machine. It is used to store all the dependencies that are needed for the project, including any dependencies that are not available in remote repositories. This allows the developer to work on the project even when there is no internet connection.

Linux: "~/m2/repository".
Windows: "C:\Users\[your username]\m2\repository"

Notes

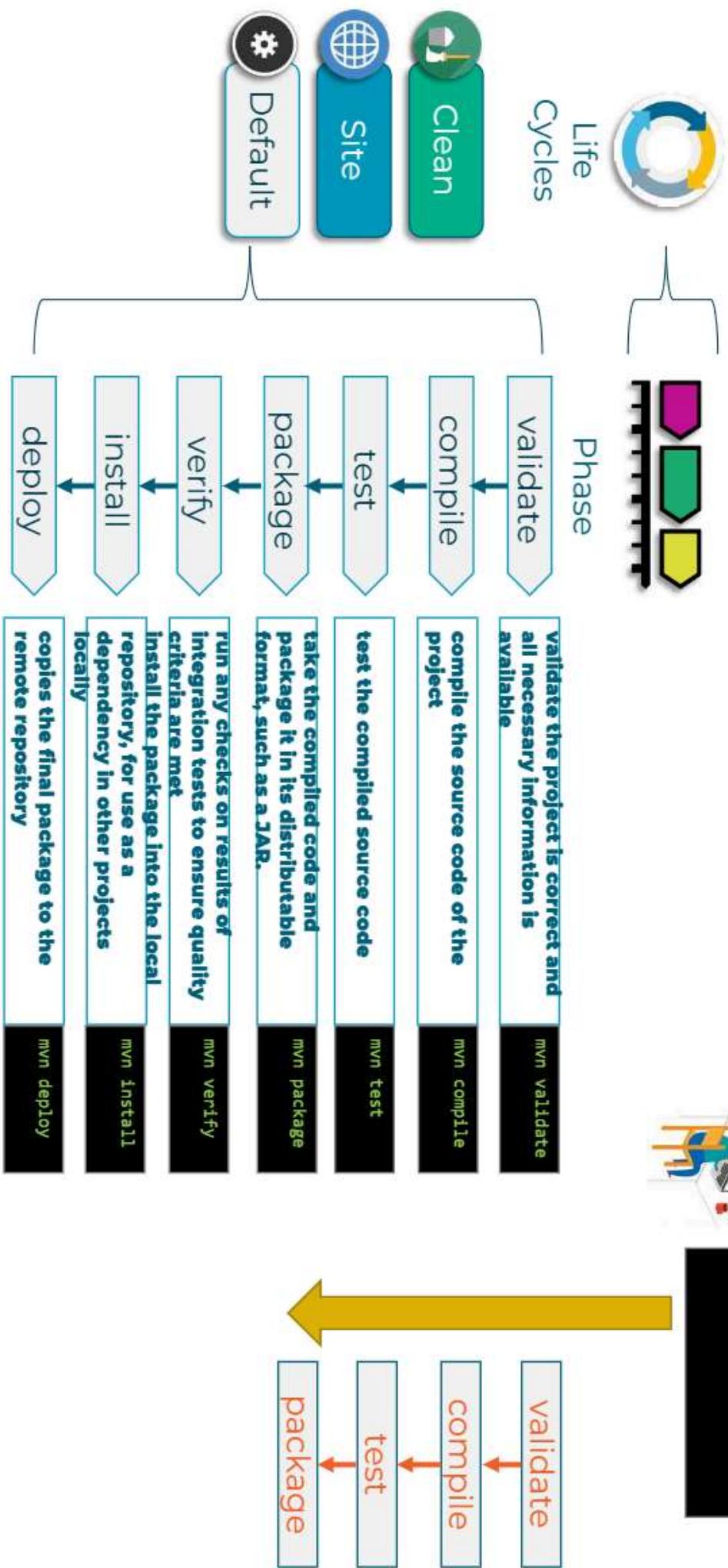
Maven works from the local repository. If a dependency is missing, it will pull from the remote repository to the local repository if it is not already in the local repository



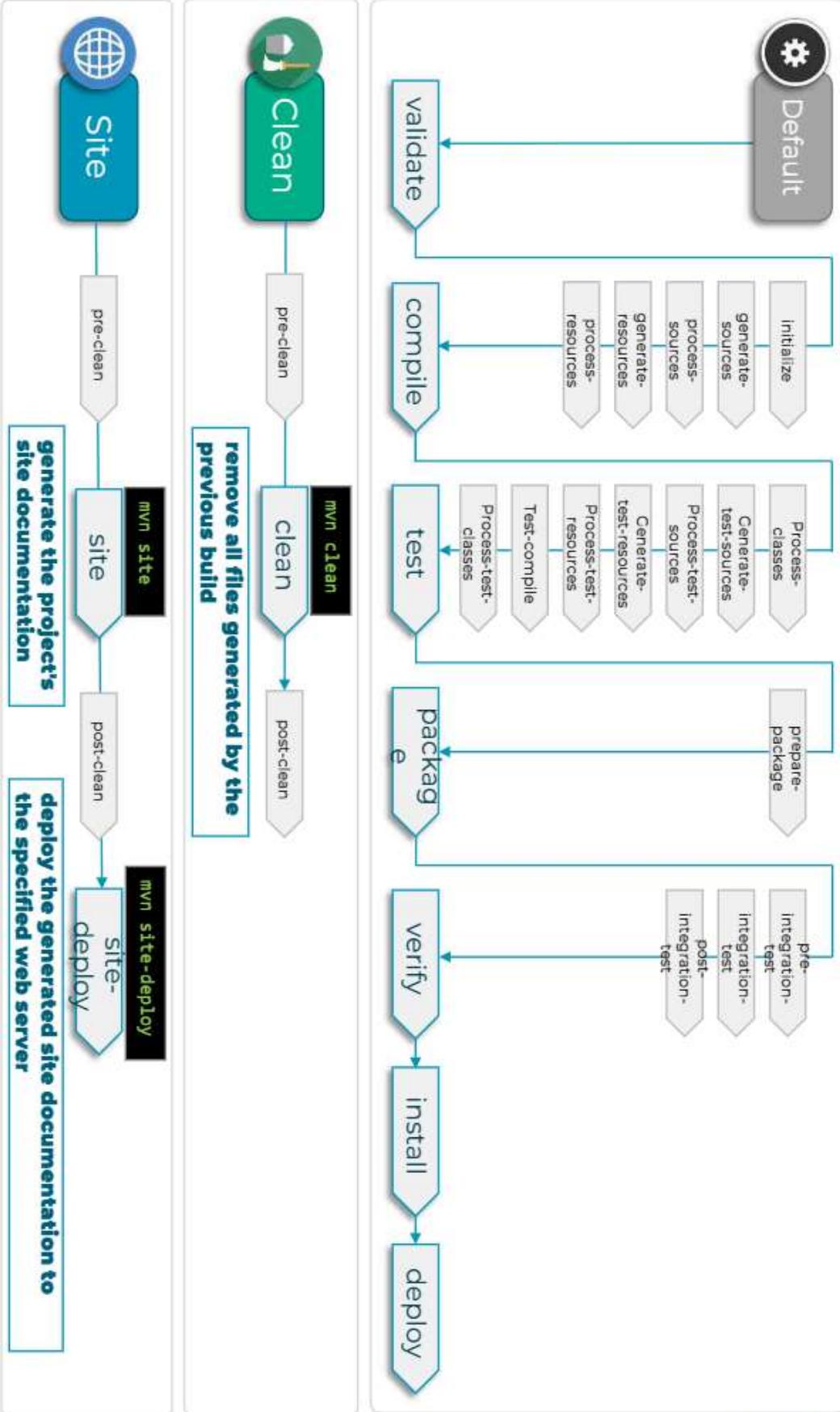
QA Life cycle and phases

Alice

> mvn package



QA The default, clean, and site life cycles



Notes

Phases named with hyphenated-words (**pre-**, **post-**, or **process-**) are not usually directly called from the command line. The same is true for **integration-test**.

See
<https://maven.apache.org/guide/s/introduction/introduction-to-the-lifecycle.html#setting-up-your-project-to-use-the-build-lifecycle>

QA More about phases: goals and plugin



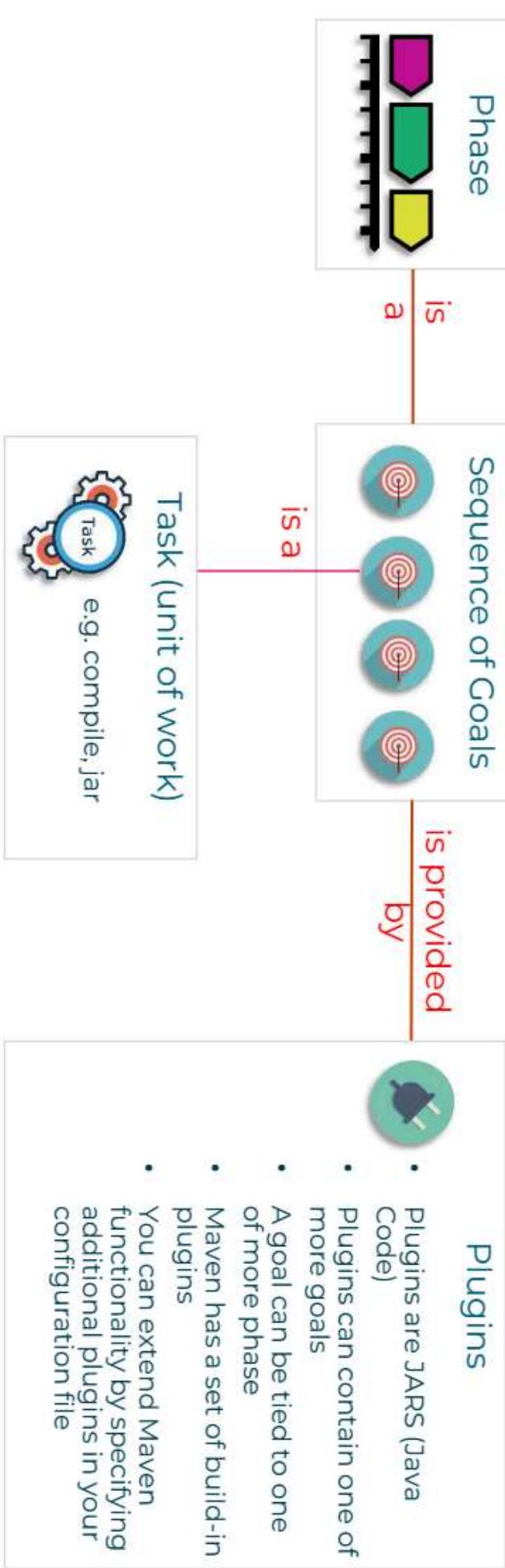
Alice

> mvn package
mvn {phase}

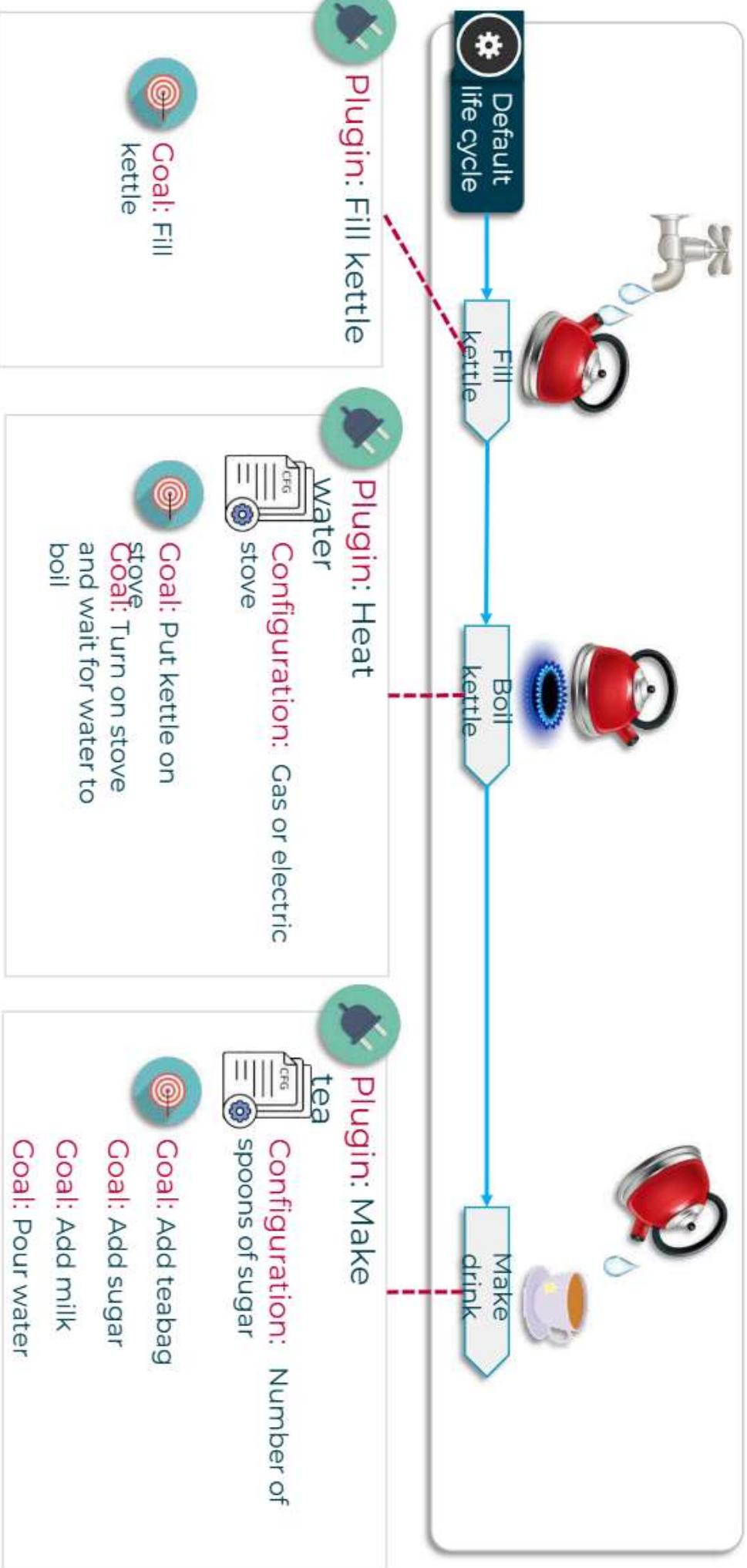


Notes

You can invoke a goal directly from a plugin but this is **not** something that developers will normally do. 99% of the time you can just work with phases.

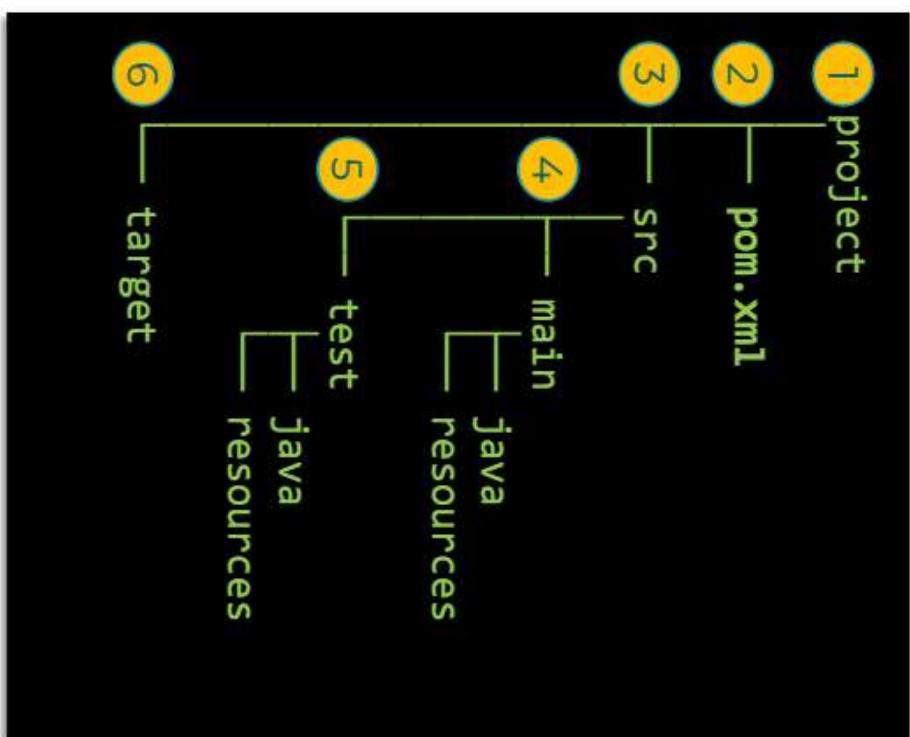


QA If Maven is for making tea





The Maven directory structure



The Maven directory structure is a standard convention which Maven expects to build a project. At the top level is the root directory of your project which can be named as required. In this case it is named as project.

At the root level of the project directory lays the pom.xml file. The POM file **must** be in the this directory and all maven commands are issue from this directory.

The **src** directory is by default where Maven looks for the application source code, test source code and any resource files.

The **target** directory contains all the final products that are the result of Maven building your project and any temporary and intermediate files needed by Maven when building your application. This directory is automatically create when you do a build.

6

target

Notes

There are other directories with special purposes that are not often used and is not needed in generally.



QA The Maven Configuration File: POM

Basic Maven POM Structure

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.example</groupId>
  <artifactId>myproject</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>
  <name>My Project</name>
  <url>http://example.com/maven/myproject</url>
<dependencies>
  <!-- Dependencies go here -->
</dependencies>
<build>
  <!-- Build configuration goes here -->
</build>
<!-- Other Elements -->
</project>
```

A Maven POM (Project Object Model) file is an XML file that contains information about the project.

<modelVersion>: Model version should always be 4.0.0.
<groupId>: This is an Id of project's group. This is generally unique amongst an organisation or a project. This similar to how Java packaging naming convention works.

<artifactId>: specifies the unique identifier for the project.

<version>: specifies the version of the project. Always 4.0.0
<packaging>: specifies the packaging type for the project (e.g., 'jar', 'war', 'ear') when used for the package phase command is invoked.

<name>: specifies the name of the project.

<url>: specifies the URL for the project's homepage.

<dependencies>: specifies the dependencies (libraries) that the project depends on.

<build>: specifies the build configuration for the project.

Q4 POM – the build element

```
<build>
<finalName>myproject</finalName>
<plugins>
  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-
    plugin</artifactId>
    <version>3.8.1</version>
    <configuration>
      <source>17</source>
      <target>17</target>
    </configuration>
  </plugin>
</plugins>
</build>
```

The `<build>` element specifies the build configuration for a project. It is used to define the following:

- The final name of the built artifact (e.g., the JAR or WAR file).

- The plugins that should be used to build the project, including their configuration.
- The build extension plugins that should be used to extend the build process.

In this example, the `<finalName>` element specifies that the built artifact should be named `myproject`.

The `<plugins>` element is where all plugin goes. Here the example specifies just one plugin, the `maven-compiler-plugin`, and that should be used to build the project, and the configuration element specifies that the source and target version of the Java compiler should be set to version 17. Each plugin needs to be in its own `<plugin>` wrapper.

QA POM – the dependency element

```
<dependencies>
  <dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-api</artifactId>
    <version>1.7.30</version>
  </dependency>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
  <version>4.13</version>
  <scope>test</scope>
</dependencies>
```

The `<dependencies>` element is used to specify the dependencies of a project. Dependencies in Maven are external libraries or modules that a project needs in order to be built, compiled, and executed. Each dependency element is wrapped in its own `<dependency>` element and is used to specify the following:

- The groupId, artifactId, and version of the dependency.
- The scope of the dependency.
- The type of the dependency (e.g., jar, war, etc.).
- The classifier of the dependency (optional).

The example here two dependency elements. The first dependency is the SLF4J API, which is a logging framework. The second dependency is JUnit, which is a testing framework.

The `<scope>` element specifies the scope of the dependency, which determines when the dependency is available to the project. In this example, the JUnit dependency has a scope of test, which means it is only available to the Test phase. If no scope is defined it is available to all phases.

QA Generating unit test report

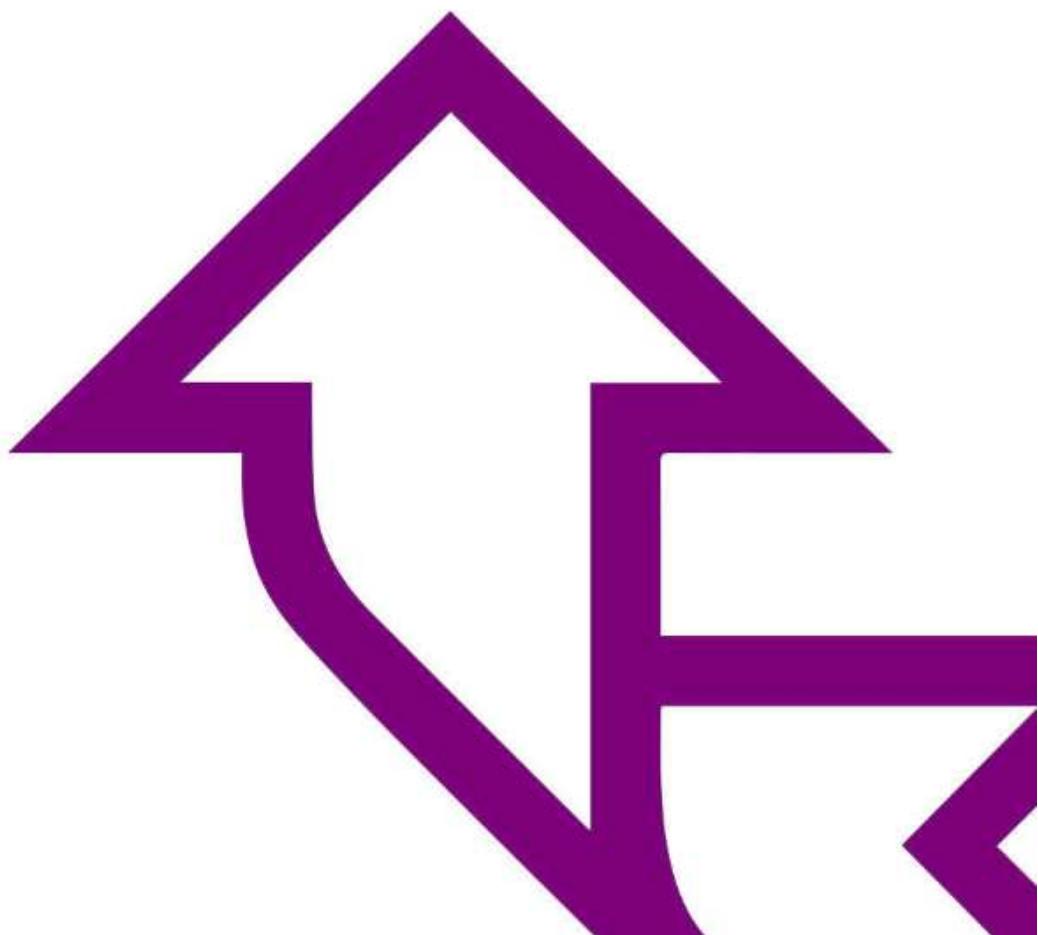
```
<reporting>
  <plugins>
    <plugin>
      <groupId>
        org.apache.maven.plugins
      </groupId>
      <artifactId>
        maven-surefire-report-plugin
      </artifactId>
    </plugin>
  </plugins>
</reporting>
```

In order to display the reports in a more human readable format you can the Surefire Reports plugin. This plugin will convert any reports generate in the Test phase from XML to HTML. The conversion can either be performed automatically via the Site goal or independently using surefire-report:report.

The report will be an HTML file called `surefire-report.html`. You can open the HTML file in a web browser to view the report.

To get a unit test report in Maven, you will need to use a plugin that generates test reports. One popular plugin for this purpose is the Surefire plugin. The Surefire plugin is used by default in Maven but to configure it you will need to add it to the build element to the POM file.

Conclusion



QA

COURSE SUMMARY

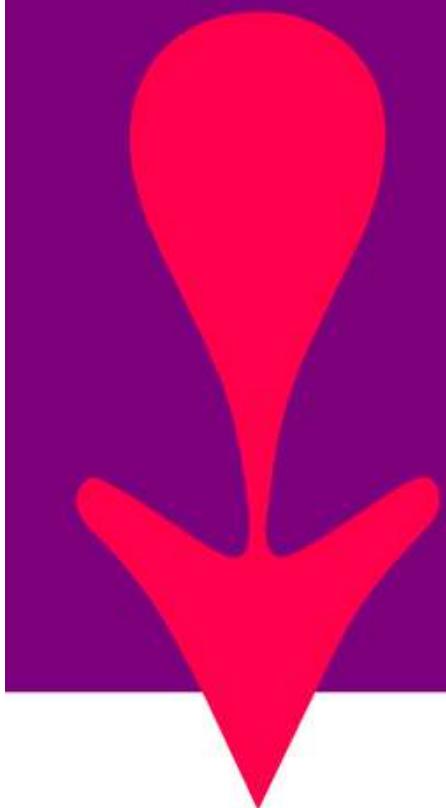


- Java and the IDE
- Primitives
- Control flow
- Introduction to objects
- Inheritance
- Interfaces
- Collections
- Error handling
- File handling
- String manipulation
- Functional programming – Lambda and streams
- Packaging it all up
- Introduction to Java modules

OUR OBJECTIVES

- 
- Develop Java code using the Eclipse IDE
 - Write programs and run them from both the command line and the IDE
 - Understand the basics of Java programming
 - Creating objects and methods
 - Control flow within methods
 - Manipulating data
 - Access control
 - Extend a superclass to create a subclass
 - Use the main collection types
 - Implement exception handling
 - Work with simple files
 - Understand the basics of functional programming in Java
 - Package up java files for distribution

FURTHER LEARNING



Books

- Cay Horstmann – Core Java
- Kathy Sierra and Bert Bates – Head First Java and exam guides

Documentation and tutorials online

- <https://docs.oracle.com/en/java/javase/11/docs/api/index.html>
- <http://docs.oracle.com/javase/tutorial/>
- <http://www.coderanch.com/forums>
- A good place to ask beginner questions

Practice!

- And practice some more, and some more

QA

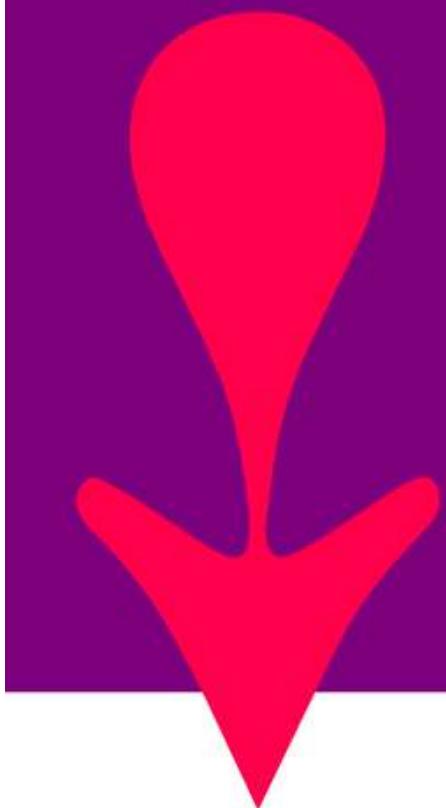
FURTHER COURSES AT QA

Java SE Programming II (OCJSEPII)

- Written by Oracle, covers more advanced concepts in Java
- Looks at Lambda and streams in more detail
- Works a lot more with modules
- Further File Handling
- JDBC
- Globalisation

QA

**WE'D LOVE
YOUR
FEEDBACK,
PLEASE**



Go to evaluation.qa.com

- Enter your course code and PIN
- Let us know what you liked and what we can improve on

vi

THANK YOU!