



Java SE

TRAINING MATERIALS - COURSE HANDOUT

Contacts

elliott.womack@qa.com

team.qac.all.trainers@qa.com

www.consulting.qa.com

Contents

Introduction	– 3
Setting up your environment	– 3
Creating & Compiling a Java File Manually	– 4
Your first class	– 5
Variables	– 5
Methods	– 6
Naming Conventions	– 8
Syntax	– 8
Flow	– 9
Primitive Data Types	– 10
Operators	– 10
Scope	– 12
The Main Method!	– 14
Conditionals	– 15
Switch Statements	– 19
Recursion/Iteration	– 20
Transfer & Control Statements – Continue & Break	– 23
Arrays	– 24
Iteration/Recursion – For each loop	– 26
Imports – Scanner	– 28
Object Orientated Programming – Intro	– 30
Pass by Reference and Pass by Value	– 47
Garbage Collection	– 49
Static	– 50
Casting	– 51
Java – Additional	– 53
Exceptions	– 56

Introduction

SETTING UP YOUR ENVIRONMENT

To get started working with Java you will need 1 thing:

- The Java Development Kit (JDK)

This has everything you need to develop java applications.

However notepad can be quite...slow...to use.

So we'll be using Eclipse mainly. (Feel free to use another IDE if you prefer)

Once you have the JDK there are a few steps you need to follow:

- 1. Install the JDK
- 2. Prepend **C:\Program Files\Java\jdk...\bin** (your path that you installed it too) to the beginning of the PATH variable
- 3. Enter `java -version` into command prompt to check that this has worked.
- 4. Install the IDE (Eclipse)

JDK, Eclipse & IntelliJ all exist in C:\LocalInstall\

If you get error **13** there is a simple fix;

Error 13 – need to edit .ini file, find the javaw.exe file in the jre folder, in the ini file, before vmargs type on a new line **-vm NEWLINE JAVA_PATH**

Java

CREATING & COMPILING A JAVA FILE MANUALLY

In notepad write this code

```
public class HelloWorld {  
    public static void main(String[] args) {  
        System.out.println(args[0]);  
    }  
}
```

Save it as **HelloWorld.java**

Open CMD

Navigate to the java file you created

Type **javac HelloWorld.java**

- This will compile your app and create a .class file

Type **java HelloWorld HelloWorld!**

- This should run your application and output "HelloWorld!"

YOUR FIRST CLASS

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

public is an access modifier, this designates how visible the class is to other classes

class is the keyword to designate that we're creating a class

HelloWorld is the name of the class we're creating

public static void main(String[] args) is the main method, this is where all execution begins.

System.out.println("Hello World!") is a function that prints out a **String** to your console, which is a data type that holds text.

Curly brackets denote scope and are required on methods and classes, as well as some other constructs.

VARIABLES

A variable is a container that holds values that we use in the program.

Every variable has a data type and a name associated with it, and eventually, a value.

A variable could be declared to use one of the eight primitive types

```
int a = 5;  
String b = "Dog";  
double c;  
c = 5.3;
```

METHODS

A method is a set of code which is referred to by name and can be called at any point in a program by simply utilising the methods name.

Think of a method as a sub-program that acts on data and often returns a value.

Every method has a name and a return type, and can also have parameters that are variables that you give it that it may need to perform its function.

```
void methodName() {  
    System.out.println("Hello!");  
}  
void methodName2(String name) {  
    System.out.println("Hello " + name);  
}  
int methodName3(int a, int b) {  
    return a+b;  
}
```

The diagram illustrates the components of a Java method signature and variable declaration. It shows the following code snippets with labels and arrows pointing to specific parts:

- Access Modifier** points to `private` in `private void exampleMethod()`.
- Return Type** points to `void` in `private void exampleMethod()`.
- Method Name** points to `exampleMethod()` in `private void exampleMethod()`.
- Variable Type** points to `int` in `int exampleInt = 3;`.
- Variable Name** points to `exampleInt` in `int exampleInt = 3;`.
- End of statement** points to the semicolon `;` in `int exampleInt = 3;`.
- Assignment operator** points to the equals sign `=` in `exampleInt = 40;`.
- Value** points to the number `40` in `exampleInt = 40;`.

Access modifier – This denotes the visibility of the method to other classes

Return Type – This signifies what type of variable the method will be returning to the caller

Method name – This is how you reference your method, should be descriptive enough to let the users know what the method does without having to look into the code.

Java

Variable Type – This is a variable declaration, a variable declaration always requires the type of the variable first.

Variable name - This is how you reference your variable, should be descriptive enough to let the user know what should be inside of it, or what it is for.

Assignment Operator – This is how we give variables data, it is “putting” the value on the right, into the variable on the left.

The diagram shows a Java method signature with several annotations pointing to its components:

- Access Modifier** points to `private`.
- Return Type** points to `int`.
- Method Name** points to `exampleMethod`.
- First parameter variable type** points to the `int` parameter.
- First parameter variable name** points to `exampleParameter`.
- Return keyword** points to the `return` statement.

```
private int exampleMethod(int exampleParameter) {  
    exampleParameter = exampleParameter + 1;  
    return exampleParameter;  
}  
  
System.out.println(exampleMethod(400));  
//will output 401
```

Parameter – Similar to a variable declaration, requires a type and a name, but does not take a value.

Parameters are the way of passing data from one method to another method, the variable name in the parameters of a method are the names you reference inside the method, they are only visible inside of that method.

Return – Whatever variable is after the return keyword will be the value returned, this value must conform to the type that the method has declared it will return. Once a return statement is hit, the method ends execution, so any code after a return statement will not be executed.

A method that declares a return type must always have a path that returns a value.

EXPLANATION OF FLOW

In the above example, we call the `println()` method, inside of that method, instead of just a string or a value to output, we're going to call the `exampleMethod()` method, which returns an integer, which will be the value that will be outputted.

Before `println()` can be executed, it needs to finish executing `exampleMethod()`, we pass 400 to the method, which then the method takes, adds one to it, then returns it. 401 is the returned value, which is then output via the `println()` command.

NAMING CONVENTIONS

Camel case – Camel case has the first letter of every word in the name capitalized, bar the first.

- camelCase, theDog, whatIsLoveBabyDontHurtMe

Camel case should be used for **variable** names and **method** names.

Pascal case – Pascal case is similar to camel case however the first letter is also capitalized.

- PascalCase, TheDog, WhatIsLoveBabyDontHurtMe

Pascal case should be used for class names

SYNTAX

{ } – These are used to surround code blocks. This includes Classes, Methods, Conditionals and Loops.

; – These are used to end a line of code

// – used to comment a line

(Comments are lines of code that are ignored by compilers that aid your code in readability, often explaining chunks of code)

/* */ – Used to surround comment blocks

/ **/** – Used to surround documentation comment blocks (Like JavaDoc)

() – Used to surround parameters and arguments

. – Used to access a variables methods and attributes.

FLOW

Code is executed with a certain flow to it, things will generally only be done one at a time, proceeding each other from top to bottom, following the flow of your program is important. If a method is executed, it will finish executing that method before executing the next line of code after that method call.

```
public static void main(String[] args)
{
    method1();
    method2();
    System.out.println(method3());
}
static void method1() {
    System.out.print("Hello");
}
static void method2() {
    System.out.print("World");
}
static String method3() {
    return "!";
}
```

FLOW EXPLANATION OF EXAMPLE

- 1. We begin at the main method, as this is where all execution starts
- 2. method1() is then called
- 3. method1 prints out "Hello"
 - › a. Note: print() is different to println(), print does not print a new line of text.
- 4. Once method1 has finished executing, method2 is the next method to execute.
- 5. method2 prints out " World"
- 6. We then call the println() command, but inside of that is a call to method3()
- 7. method3() is then executed **before** println() is finished executing
- 8. method3() **returns** "!"
- 9. println() can then finish executing, the value that it then prints is what was returned from **method3()**, which was "!"
- 10. The final output will be **Hello World!**

Java

PRIMITIVE DATA TYPES

Primitive data types are what java is built upon, almost the base level of data that java can work with.

Data Type	Representation	Range	Default Value
<u>boolean</u>	n/a	true or false	false
byte	8	-128 to +127	0
char	Unicode	\u0000 to \uFFFF	\u0000
short	16 bit	-32768 to 32767	0
<u>int</u>	32 bit	-2147483648 to 2147483647	0
long	64 bit	-922337206854775808 to 922337206854775807	0L
float	32 bit	3.4e +/- 38(7 digits)	0.0f
double	64	1.7e +/- 308(15 digits)	0.0d

OPERATORS

ASSIGNMENT

- =

ARITHMETIC

- + Plus
- - Minus
- / Divide
- * Multiply
- % Modulus

UNARY

- + Indicates a positive value (Numbers are positive without it, too)
- - Indicates a negative value.
- ++ Increments by 1
- -- Decrements by 1
- ! The not operator - Inverts the value of a boolean

Java

EXAMPLES

```
int num = 0;
num = num + 1;
num = num - 14;
num = num % 29;
num = num / 900;
num = num * 34222;
num++;
num--;
num += num;
num -= 4;
num %= 232;
num /= 452;
num *= 1;
boolean b = true;
b = !b;
```

SCOPE

Scope refers to the visibility of aspects of your program.

The rule of thumb for java is every time you use `{ }` it creates a new level of scope, there are a lot of situations where you have to use brackets for certain constructs.

There are three main levels of scope in Java.

CLASS LEVEL/INSTANCE SCOPE

Variables that are referenceable throughout the entire class, these methods are inside the class but outside of methods.

Generally defined at the top of the class.

METHOD/LOCAL SCOPE

Variables that are temporary and (generally) only used in the method they are declared in.

As soon as the method ends all variables declared inside that method are no longer referenced too and cannot be accessed any more.

LOOP SCOPE

Variables that are declared inside a loop declaration, only accessible inside the loop and are lost after the loop is ended.

Java

EXAMPLE

```
public class HelloWorld {  
    // not valid  
    int n = n1;  
    // valid  
    int n0 = 0;  
  
    public void scope() {  
        // valid  
        int n1 = n0;  
  
        { // valid  
            int n2 = n1;  
        }  
    }  
    // not valid  
    int n3 = n2;  
}  
}
```

THE MAIN METHOD!

The main method is a very important part of a Java SE application.

It is where execution starts, and ends.

The format for the main method is **very** specific, and can't really be changed.

If the format for your main method is not correct you may have troubles running your project.

SYNTAX

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

public – This is required so that when the JVM attempts to run your file, the main method is visible to it.

static – This is required as without static it implies that you need to create an instance of the class the method is contained in, to use it. Which wouldn't be helpful for the main method as it doesn't belong to any class/object

void – The main method shouldn't ever return a value.

String[] args – The main method has to have a way of accepting arguments that may be provided at runtime.

CONDITIONALS

Conditional operators return a boolean value as a result of the operator statement.

A common conditional operator is `==`, this returns true or false depending on if the two variables are equal to each other.

```
int a = 1;
int b = 1;
a==b // returns true
```

OPERATORS

Equality

- `==` Is equal too
- `!=` Not equal too
- `<` Less than
- `>` Greater than
- `<=` Less than or equal too
- `>=` Greater than or equal too

Type Comparison

- `instanceof` – compares an object to a specified type

You can also group conditional statements together with `&&` and `||` which are **and** and **or** respectively.

```
Int a = 1;
Int b = 2;
a > b && b > 1 // returns false, a is not greater than b
even if b is greater than 2, both have to return true for
the entire statement to be true.
```

Java

OPERATORS

- `&&` and
- `||` or
- `&` and
- `|` or

The difference between double operators and single operators is that the double operators will only evaluate the second statement if it needs to, however the single operator will always evaluate the second statement even if it's irrelevant.

e.g. In an and statement, if the first statement is false, it doesn't need to evaluate the second statement since it doesn't matter if its true or not, the total result will be false.

In an or statement, if the first one is true, the second one doesn't need to be evaluated.

CONDITIONAL USAGE – IF/ELSE

Conditionals are generally used with `if` and `else if`'s.

An if statement is a construct that requires a Boolean statement to “check”.

If the Boolean statement is true, it will run some specified code.

If the Boolean statement is false, it will not run the code, effectively skipping it.

Syntax

```
if(statement) {  
  //code  
}
```

If statements can be immediately followed by an `else` statement, which is executed if the `if` statement does not execute, almost like the alternative.

Java

Syntax

```
if(statement)
{
    //code
}
else{
    //alternative code
}
```

If statements can also be followed by an **else if** statement, which is executed if the previous if statement did not execute, and can have its own condition to evaluate. else ifs can also follow other else ifs.

Syntax

```
if(statement) {
}
else if(anotherStatment) {
}
else{
}
```

An else statement, if used, must be immediately after an if or an else if. You cannot have an else if after an **else**

EXAMPLES

```
boolean isDevCool = true;
if (isDevCool) {
    System.out.println("Dev is Cool");
} else {
    System.out.println("Dev isn't cool.");
}

int devsMoney = 3;
boolean isDevDrunk = true;
if (devsMoney > 0 && isDevDrunk) {
    System.out.println("He will buy you drinks");
} else {
    System.out.println("He won't buy you drinks");
}

int devsMoney = 90;
boolean devCanPlaySmash = false;
if (devsMoney > 5 && devCanPlaySmash) {
    System.out.println("Dev enters a smash tournament");
} else if (devsMoney < 1 && devCanPlaySmash) {
    System.out.println("Dev is just too poor to enter");
} else {
    System.out.println("Dev just cant play smash");
}
```

SWITCH STATEMENTS

Switch statements are a useful way of checking what a variable is against numerous cases, if you had one variable and you wanted to check if it was 1 of 20 things, you might end up with 20 if statements to do so, Switch statements are a better way of doing so.

SYNTAX

```
switch(variableYourComparing) {  
    case whatYourComparingItTo:  
        //code to execute  
        break;  
    case whatElseYourComparingItTo:  
        //code to execute  
        break;  
}
```

Breaks are **not** required, however if you don't have a break statement then the code will "fall through" and execute everything from every case below it. Which may not be desirable. Switch statements can also contain a **default** case, which is executed if it can't match it against anything you've provided.

Example

```
int day = 3;  
switch (day) {  
    case 1:  
        System.out.println("Garfield hates Mondays");  
        break;  
        // etc, the other days of the week  
    case 6:  
        System.out.println("Saturday");  
        break;  
    case 7:  
        System.out.println("Sunday");  
        break;  
    default:  
        System.out.println("Not a day");  
        break;  
}
```

RECURSION/ITERATION

Recursion is the ability to execute code multiple times in a tidy format. Since if you're doing a similar thing multiple times, you don't want to have a line of code for every time you are doing that

FOR LOOP

A for loop is used when you know how many times you want to iterate.

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

The diagram illustrates the components of a for loop. Three orange arrows point down from the labels 'Initialisation', 'Exit condition', and 'Update' to the corresponding parts of the loop header: 'int i = 0', 'i < 10', and 'i++' respectively. A fourth orange arrow points left from the label 'Processing' to the body of the loop, 'System.out.println();'.

The initialisation is the variable that the loop is going to work with, we can also set what that variable will start at.

The exit condition is a Boolean statement that signifies “when it is going to keep going”, so in this case the loop is going to keep executing as long as *i* is less than 10.

The update is what code is executed at each iteration, here we are going to increment what *i* is by one each loop.

Then the code in the brackets is the main body of the code that we will execute every iteration.

In this example the loop will execute 10 times, going from 0 to 9, then stopping.

Example

```
for(int i = 3; i < 6; i++){  
    System.out.println(i);  
}
```

This example would print out 3,4,5.

Java

WHILE LOOP

A while loop is used when you don't know when it's going to stop, or may have multiple reasons to stop. It takes a single Boolean condition and will keep iterating until that condition becomes false.

Syntax

```
while(statement) {  
    //do code  
}
```

Example

```
int catCount = 0;  
boolean notEnoughCats = true;  
while (notEnoughCats) {  
    System.out.println("Another cat");  
    catCount++;  
    if (catCount > 273)  
        notEnoughCats = false;  
}  
System.out.println("Gareth has too many cats");
```

DO WHILE

A do while loop is very similar to a while loop, however it always does one iteration before even checking the condition.

Syntax

```
do {  
    //code  
} while(statement);
```

Java

Example

```
int playCount = 0;
boolean playing = true;
do {
    System.out.println("Playing");
    playCount++;
    //can only play 10 times
    if (playCount > 10)
        playing = false;
    //or if you die
    if (!isAlive())
        playing = false;
} while (playing);
System.out.println("Game Over!");
```

While – Realistic Example

```
Scanner sc = new Scanner(System.in);
boolean badInput = true;
int input = 0;
do {
    System.out.println("Please enter a number");
    try {
        input = Integer.parseInt(sc.nextLine());
        badInput = false;
    } catch (Exception ex) {
        System.out.println("Please enter a valid number");
    }
} while (badInput);
System.out.println("Your number was: " + input);
```

- Will loop and ask for user input until they input something that parseInt can successfully take, aka when they put in an integer.
- Otherwise it will keep asking them for input.
- It's a do While because we want to ask them for input at least once.
- It will escape the loop once it gets past the parseInt line and sets badInput to false, meaning on the next conditional check it will not pass and exit the loop.

TRANSFER & CONTROL STATEMENTS – CONTINUE & BREAK

Continue and break are statements that allow us to “control” the outer loop from within.

Break breaks the loop, exiting out of it.

Continue skips to the next iteration, not executing any code after the statement.

Example

```
for (int i = 0; i < 10; i++) {  
    if (i == 7)  
        break;  
    if (i == 2)  
        continue;  
    System.out.println(i);  
}
```

This would output 0,1,3,4,5,6

ARRAYS

Arrays are used for holding a collection of data. There are numerous different types of arrays and Collections that are intended for different situations.

A basic array is similar to a variable declaration, however it takes a bit more syntax

SYNTAX

Declaration must conform to the following, simply adding square brackets to a normal variable declaration

```
int[] arrayOfInts
```

The right hand side of the declaration, the object which we'll be creating, can be two different ones.

In the first way, we create an array of a certain size that will be initially empty.

```
int[] arrayOfInts = new int[5];
```

In the second way, we create an array with values already inside of it.

```
int[] arrayOfInts = {1,2,3,4,5};
```

Once an array is created you can access the elements inside the array via the square bracket notation, passing in the index of the array you'd access.

Note: Array indexes start at 0, so for a 3 element array, the first element is at index 0 and the third element is at index 2

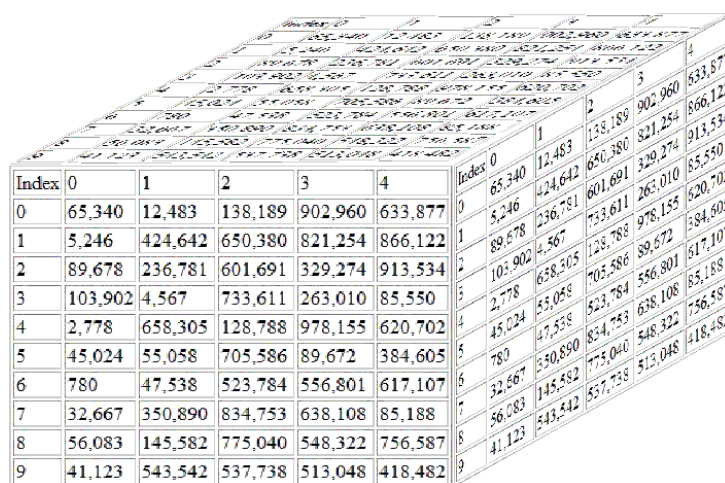
```
arrayOfInts[2]; //this would retrieve the third element in  
the array
```


Java

MULTI-DIMENSIONAL ARRAY

Arrays can also be multi-dimensional, logistically its an “array of arrays”, where the elements of the first array are arrays, which in turn have values. To visualize this you might want to think of a table, then for 3 dimensional arrays, a cube.

	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]



Index	0	1	2	3	4
0	65,340	12,483	138,189	902,960	633,877
1	5,246	424,642	650,380	821,254	866,122
2	89,678	236,781	601,691	329,274	913,534
3	103,902	4,567	733,611	263,010	85,550
4	2,778	658,305	128,788	978,155	620,702
5	45,024	55,058	705,586	89,672	384,605
6	780	47,538	523,784	556,801	617,107
7	32,667	350,890	834,753	638,108	85,188
8	56,083	145,582	775,040	548,322	756,587
9	41,123	543,542	537,738	513,048	418,482

ITERATION/RECURSION – FOR EACH LOOP

For each loops are a special “enhanced” typed of for loop that iterates through arrays/collections.

They’re a lot easier to write, read, and use.

However they don’t provide a current “iteration” count, like normal for loops do.

Example

```
int nums[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 0};  
for (int i : nums) {  
    System.out.println("Number: " + i);  
}
```

Would output 1,2,3,4,5,6,7,8,9,0.

“For every integer i, in nums, do x”

Example

```
for (int i = 0; i < votes.length; i++) {  
    process(votes[i]);  
}  
for (int vote : votes) {  
    process(vote);  
}
```

Both of these loops do the same thing, however the first loop has the benefit of access to a variable that tracks what iteration it’s currently on, as well as the ability to change where the loop starts, ends, and how it increments.

The second one has the advantage that it’s a lot simpler to read and write, so if you don’t need the increased functionality the former loop provides, the latter is better.

Java

For

```
int twoDArray[][] = {{0, 1, 2}, {1, 2, 3}, {2, 3, 4}};
// columns
for (int i = 0; i < twoDArray.length; i++) {
    // rows
    for (int j = 0; j < twoDArray[i].length; j++) {
        System.out.print(twoDArray[i][j]);
    }
    // after each row, print a new line
    System.out.println();
}
```

Foreach

```
int twoDArray[][] = {{0, 1, 2}, {1, 2, 3}, {2, 3, 4}};
// columns
for (int[] a : twoDArray) {
    // rows
    for (int b : a) {
        System.out.print(b);
    }
    System.out.println();
}
```

Both of these examples do the same thing. Outputting;

```
012
123
234
```

IMPORTS – SCANNER

Imports are a way of providing access to other sets of code that you might want access to. You don't inherently have access to everything that java has to offer as that would make java applications **very** bulky, with a lot of code that isn't being used.

So the idea is to just import things that you are going to use.

Syntax

```
import package/class name
```

Example

```
import java.util.Scanner;
```

`java.util` is a package that contains classes, and `Scanner` is the class that we want to import, so we can use it.

SCANNER

The scanner class is one of many ways of taking input off a user, what you might use to create CLI (Command line interface) or manually test your code with various inputs that you enter.

There are two parts to use the Scanner class, first we have to create the Scanner object, and pass it a stream to "scan"

```
Scanner sc = new Scanner(System.in);
```

Scanner is the name of the class we're creating an object from, `sc` is the name of the variable (how we refer it), the **new** keyword signifies we're creating an object via the classes blueprint, `Scanner` again is saying we're creating an object that is a **Scanner**, then in the parenthesis we need to pass a **Stream**, in this case we're passing **System.in** which is our command line input.

So this in total reads "Create a Scanner object, with the variable name `sc`, that is attached to the `System.in` stream, which is our console input.

Java

Once the object is created, we can call methods within that object, which is going to be necessary if we want our object to do anything.

```
sc.next();
```

This will read the next “word” from the input, so if you entered “Jeff” it would return “Jeff”, if you entered “Jeff Smith” it would still only return jeff, since Jeff is the first word.

There are numerous methods like next(), have a look at the official documentation of the Scanner class on oracles website

<https://docs.oracle.com/javase/7/docs/api/java/util/Scanner.html>

(this is good practice for any class you are not sure how it works/wondering if something exists that helps you out!)

OBJECT ORIENTATED PROGRAMMING – INTRO

Object Oriented Programming is the current ‘flavour’ of programming. There are many principles behind OOP and this section will give you an insight into the 4 key ones.

OO Means, at its core, the ability to define your own (complex) types. Or your own “Objects” that your code will then work with.

An Object is a single component of a system.

For example in a car park there will be barriers at the entrance and exit, a ticket dispenser next to every entrance barrier, a ticket pay station and a counter for the number of spaces left.

Each of these can be described as objects in the ‘system’ of a car park.

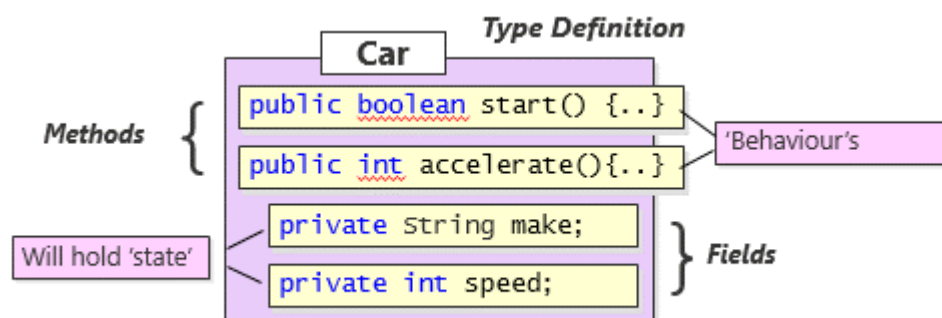
The easiest way to visualise this is by using a class diagram to identify all the potential classes.

OO DATA TYPE

Definition of a data type is a single entity

Fields/Attributes - The DNA of your object

Methods - Functions that define behaviour in your object.



Java

Attributes

Attributes are the DNA of your object.

As your DNA controls your hair, eye and skin colour as well as a number of other factors, the Attributes of an object are used to store information related to the class.

Methods/Functions

Methods are the way the class does things, they are its functions.

For example when you press the '+' button on a calculator you are calling the calculators sum method that will perform the action and then return the result.

Constructors

Constructors are like the blueprints of your class. Whenever you create a new instance of a class its constructor is called.

The constructor is used to set up any attributes that are required to be set up either with default values or with values that are passed in.

ATTRIBUTES

When adding attributes you need to consider what the object needs to know about itself in order to carry out its purpose.

This is specifically relating to the state of the object.

If we think back to our car park, a barrier may need to know whether it is raised or lowered as this relates to the state of the barrier.

Attributes can be primitive type, or even other Objects that you create!

Think about if you wanted to model a Person, and in your system you wanted to have an attribute such as "Pet", this could be null if they didn't have a pet, but otherwise it could be an object such as Dog !

Java

METHODS

Objects can have multiple Methods.

For example a barrier will have a method to raise the barrier and another one to close it where the `raiseBarrier()` method may tell the program that a car has entered while the `lowerBarrier()` method may tell the program that a car has left the car park.

You may find when thinking about your methods that you realise you may have missed some attributes

CONSTRUCTORS

When specifying a constructor we need to consider our attributes and think about which of them need to be set up when we create an instance of the class.

Attributes which may need to be set up differently depending on the context will need to be set in the constructor.

Attributes that have an initial value that is not dependent on the context do not need to be set in the constructor.

A default constructor () will be provided by the compiler if you don't write one.

Constructors can be overloaded and chained together.

This method is called when the "new" keyword is used to create an object.

Constructors have no return type, and have to be the same name as the class.

Example

```
public class Barrier {  
    public Barrier(boolean isRaised) {...}  
}  
Barrier b = new Barrier(true);
```


Java

KEYWORDS – THIS

this refers to the object on which method was invoked

- In an ‘instance’ context there is always a ‘this’ *****
 - › It is a reference to the object on which method was invoked
 - » Think of it as a ‘hidden’ 1st parameter (of each instance method)

Example

```
public class Barrier {
    public boolean isRaised;
    public void raiseBarrier() {
        this.isRaised = true;
    }
    . . .
}
```

ENCAPSULATION

Types encapsulate state and “secret” behaviour

- Introduces loose coupling between code
- Limit the visibility of an objects attributes or methods
- Java controls accessibility using access modifiers
- Apply to type definitions - the class itself
- But also to its members - fields and methods
- The goal is to be able to control what can directly manipulate attributes and methods.

There are four access modifiers that enable encapsulation

Public		Protected		Default		Private	
Public means that attributes and methods can be seen and accessed without any restrictions		Protected means that attributes and methods can't be seen or accessed without inheritance if the object is not in the same package		Default is the access level that is applied to attributes and methods if no modifier is specified. Objects must be in the same package		Private means that attributes and methods can only be seen and accessed from within the same object	
Class:	Y	Class:	Y	Class:	Y	Class:	Y
Package:	Y	Package:	Y	Package:	Y	Package:	N
Subclass:	Y	Subclass:	Y	Subclass:	N	Subclass:	N
World:	Y	World:	N	World:	N	World:	N

Java

Getter Methods

A getXxxx() method will not be void

- Will rarely receive a parameter
- Typically used to encapsulate and make available a private field
- Often one line of code
- Client code will use the value returned – somehow
 - › Unlikely to be calling the method just for its ‘side-effects’

A get method can be used to perform a calculation referring to one or more fields

Example

```
public class Barrier {
    private boolean isRaised;

    public boolean getisRaised() {
        return this.isRaised;
    }
}

public class Car {
    private int speed;

    public int getSpeedKPH() {
        return speed * 8 / 5;
    }
}
```

Setter Methods

‘set’ methods are usually void

- But nearly always take a parameter used to change the state of a field
- Later.. they might throw an ‘exception’ when they can’t perform action
- If you were authoring class Car would you name a method
- **setGear(int gear)** or **selectGear(int gear)** – it is still a ‘set’ter.

Java

Example

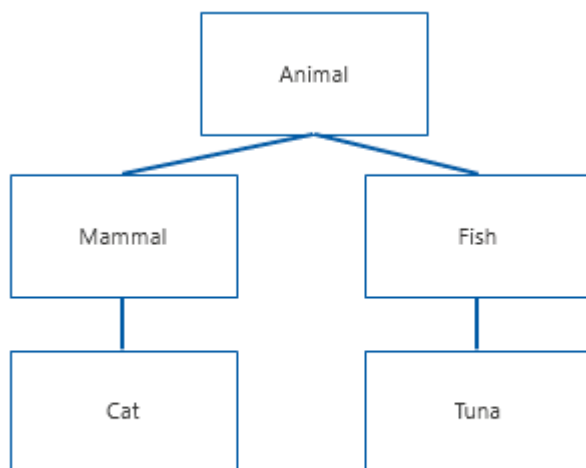
```
public class Barrier {  
    private boolean isRaised;  
    public void setisRaised(boolean isRaised)  
    {  
        this.isRaised = isRaised;  
    }  
}
```

INHERITANCE

Inheritance is used to provide classes with some common functionality by inheriting from an object that already contains that functionality. This improves efficiency by reducing code duplication.

There are 3 ways we can do inheritance:

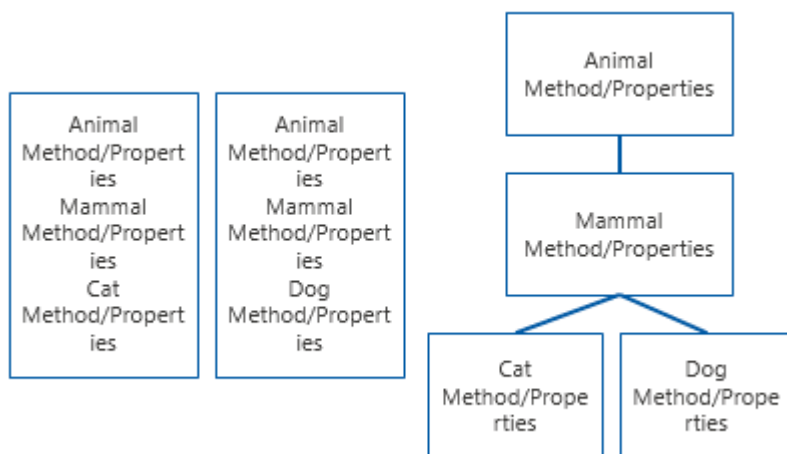
- By inheriting from an existing class.
- By inheriting from an Abstract Class.
- By inheriting from one or more interfaces.
- Cat inheriting from Mammal would make Mammal the Super/Parent class and Cat the Sub/Child class



A Cat IS A Mammal which IS A Animal

Java

- Say we want to have a Cat class and a Dog class, both need to represent the properties and methods that belong to a mammal, and an animal.
- So the Dog and Cat class both need to have all this code.
- One way is to create the class and insert all of that functionality inside of it.
- Another way is to create a hierarchy of inheritance
- This lets us re-use code and overall write less, making our actual code much more readable too.



Abstract Classes

Abstract classes are classes that cannot be instantiated so they must be implemented.

Abstract classes can contain fully written code and abstract methods.

Abstract methods must be implemented by any class that inherits from the abstract class.

So whenever we inherit this Animal class, that class has to then create

Java

Example

```
public abstract Animal {  
  
    public boolean eat(Food food) {  
        //do something with food  
    }  
    public abstract void sleep();  
    public abstract void makeNoise();  
  
}  
  
public class Cat extends Animal {  
  
    public void sleep(){  
        //sleep like a cat  
    }  
    public void makeNoise(){  
        meow();  
    }  
  
}
```

Explanation

In this example, every animal might eat in the same way (Yes some might use a mouth, some might not...) But given food, an animal can consume it for nutrients etc.

However every animal might sleep in a different way, so we'll make this method abstract (to be implemented further down the chain)

And the same with make noise.

Java

INTERFACE CLASSES

Interfaces cannot contain full code, only method stubs and final variables. Every method is inherently abstract and public.

They are useful because whereas classes can only extend one class they can implement multiple classes

List interfaces after the base class (if any) via keyword implements

- All members must be implemented

Example

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

public interface Renderable {
    void draw();
}

public class Rectangle extends Shape implements Renderable {
    private int height;
    private int width;
    public void draw() {
        ...
    }
    public float getArea() {
        get { return height * width; }
    }
}
```

Explanation

Now every shape we create (Such as rectangle) doesn't inherently need to have a draw() function, however we have the option of implementing that draw function by making the shape Renderable via implementing that appropriate interface class.

Java

Example

```
interface GearBox {  
    void changeGear();  
}  
  
class ManualCar implements GearBox {  
    public void changeGear(){  
        //change gear manually  
    }  
}  
  
class AutomaticCar implements GearBox {  
    public void changeGear(){  
        //change gear automatically  
    }  
}
```

Explanation

So every car has a gearbox, whilst some have automatic working ones and some have manual ones, they both need a way to change gears yet they do them differently, hence why we can use an interface to do so.

Java

INTERFACES

Surely i could just write the methods if they're appropriate?

Why do I need to create a separate class to then tell me I need to write them?

Isn't this the opposite of code re-use?

- **Yes & No** – Think of an electric outlet.
- The interface for an electric outlet is exactly the same, 3 pins. (not strictly true but shh.)
- Imagine a hypothetical interface called Pluggable , that means that it has a standard set of prongs that will plug into a standard outlet.
- A Dog does not implement the Pluggable interface, so we know we cant plug a dog into an electrical outlet.
- However a PS3 does implement the Pluggable interface, so we know we can plug it in.
- 1. Encourages smart application design.
- 2. Promotes Readability
- 3. Promotes Maintainability
- 4. Allows flexibility

Coding to an interface is where you set the reference of a **variable** to the Interface but the actual object is the class itself.

Such as `List<String> tmpList = new ArrayList<String>();`

ArrayList implements the list interface, so our object is ArrayList but the reference is List.

- One of the major benefits of using interfaces is how you can link multiple seamlessly unrelated objects together with them
- Take the two objects Apple and Pen. Nothing really related about them, can't have a common parent class or anything. (Like Animal was)
- If we wanted to create a method that performs a similar function on both of them, we'd have to have one for each type.
- But with an interface, that could be the common factor!

Java

Example - Before

```
class Apple {  
    double weight = 40.3;  
    public void output() {  
        System.out.println(weight);  
    }  
}  
  
class Pen {  
    String brand = "Byro";  
    public void output() {  
        System.out.println(brand);  
    }  
}
```

Example – After

```
interface Info {  
    public void output();  
}  
  
class Apple implements Info {  
    double weight = 40.3;  
    public void output() {  
        System.out.println(weight);  
    }  
}  
  
class Pen implements Info {  
    String brand = "Byro";  
    public void output() {  
        System.out.println(brand);  
    }  
}
```

Java

Not much different right? But wait, there's more!

We can now code to an interface, storing our apple object in an Info reference!

```
Info i = new Apple();  
i.output();
```

Which allows us to do this, very nice and tidy, without this we would need a separate method for every type of variable we'd want to call that method on!

```
public static void main(String[] args)  
{  
    Apple a = new Apple();  
    Pen p = new Pen();  
    outputInfo(a);  
    outputInfo(p);  
}  
private static void outputInfo(Info i) {  
    i.output();  
}
```

POLYMORPHISM – OVERLOADING

It is possible to write multiple methods in one class with the same name. This is called method overloading.

The way you do this is by specifying different parameters for the method.

With constructors you can chain overloaded methods by calling 'this()' with the required parameters.

If we write "multiply(...)" and pass it the parameters of one integer, the first method will be called, if we call it with two integers, the second one will be called.

Example

```
public int multiply(int value) {  
    return this.multiply(value, 5);  
}  
public int multiply(int value, int multiplier) {  
    return (value * multiplier);  
}
```

Java

POLYMORPHISM – OVERRIDING

Similar to our animal example, every human will have a method to eat.

Boy extends from Human, bringing over the eat method from Human

However we don't want Boy to use the Human eat method, we want it to do it in its own way.

So we create another eat method in Boy, this overrides the eat method in Human.

Example

```
class Human {  
    public void eat() {  
        System.out.println("Human is eating");  
    }  
}  
  
class Boy extends Human {  
    public void eat() {  
        System.out.println("Boy is eating");  
    }  
}
```

USING OBJECTS

Creating an object is just like any other variable, but the value of a normal instantiation is slightly different.

The normal statement for a variable like so goes in a specific fashion

Type name assignment value

```
int a = 5;
```

An object requires slightly more

```
Object a = new Object();
```

Java

We require the new keyword to signify we're creating a new object, and we need the parenthesis around the value as that's going to invoke a constructor of the class, if it parenthesis are empty then it will call a constructor that is also empty, if it's got a value(s) in, then it will again try and find a constructor that matches a value you're providing.

FULL CLASS EXAMPLE

```
public class Dog {  
  
    public int age;  
    public String name;  
  
    public Dog(int age, String name) {  
        super();  
        this.age = age;  
        this.name = name;  
    }  
    public void makeNoise() {  
        System.out.println("woof");  
    }  
    public String toString() {  
        return "Age: " + age + " Name: " + name;  
    }  
}
```

Java

THE OBJECT SUPERCLASS

Every object you create, or use, inherits from the superclass Object

This means that every object inherits a set of methods already defined in the Object superclass, all of which can be overridden.

- 1. `clone()`
- 2. `getClass()`
- 3. `equals()`
- 4. `finalize()`
- 5. `hashCode()`
- 6. `notify()`
- 7. `notifyAll()`
- 8. `toString()`
- 9. `Wait()`

LIST IMPLEMENTATIONS

- The arrays shown previously have been primitive arrays.
- Seeing as Java is an Object orientated language you will most likely need arrays that can store Objects.
- The most common array you will use will be the ArrayList:

```
ArrayList<Object> objects = new ArrayList<>();
```

- There are many other types of array such as the Map, Set & List which are each tailored to different scenarios.
- The **List** interface extends **Collection**.
- Precise control over where in the list each element is inserted.
- User can access elements by their integer index or via searching for elements in the list.
- Provides a special iterator called **ListIterator**

Java

- Some **List** implementations prohibit null elements, some have restrictions on their type of elements
- `LinkedList` is another commonly used implementation of `List`
- **ArrayList** – Index based system
- **Linked** list – Doubly Linked list system
- Because of these systems they implement they have their own strengths.
- **ArrayList** – Faster searching/Lower overhead
- **Linked list** – Faster deletion/Insertion
- `List` is an ordered `Collection`

ListIterator allows for element insertion and replacement, bidirectional access in addition to the normal operations that the `Iterator` interface provides.

The searching functions that `List` provides are extremely inefficient and should be used with caution.

Don't assume just because `List` allows null elements that an implementation of `List` will as well.

ArrayList maintains an index based system for its elements which implicitly makes it faster for searching for an element in the list.

LinkedList implements a doubly linked list system which requires the traversal through all the elements to search for an element.

LinkedList stores extra information (positions/neighbour nodes) – High memory consumption.

LinkedList deletion only requires the change of the neighbour node pointers whilst array list requires all data to be shifted to fill the space.

PASS BY REFERENCE AND PASS BY VALUE

- **Pass by value** means the called functions' parameter will be a copy of the callers' passed argument.
- So if you call a method with an integer value of 5, the parameter that "takes" that integer will copy that value to then be used.
- Think of it like this:
 - › I have a bucket of paint, I **tell** you that 5 litres are in it, you now have the knowledge that there are 5 litres of paint in my bucket.
- So if you change that 5 to a 3, I still have five, but you're telling me I have three, you're not actually changing my amount of paint.
- **Pass by reference** means the called functions' parameter will be the same as the callers' passed argument, (Not the value of the parameter, but the actual variable itself)
- Every variable has a location in memory where it is stored, passing by reference means that you're giving that method access to the actual variable.
- Think of it like this:
 - › I have a bucket of paint, I **give** you permission to use my bucket of paint, and then you can determine that there are 5 litres of paint in it.
- Now that you have permission to use my paint, if you change it from 5 to 3 (pouring it out...) then I actually have 2 less litres in my bucket

Java is pass by value

In the Java Language Specification 8.4.1 Formal Parameters section it states:

"When the method or constructor is invoked , the values of the actual argument expressions initialize newly created parameter variables, each of the declared type, before execution of the body of the method or constructor."

- When an object is passed as an argument, that object is not literally passed as an argument to the method. Internally that objects reference is passed by value and it becomes a formal parameter in the method
- Java passes object references by value

Java

Example

```
public static void main(String[] args)
{
    Person p1 = new Person("Elliott");
    System.out.println("Before swap p1: " + p1.name);
    modify(p1);
    System.out.println("After swap p1: " + p1.name);
}

public static void modify(Person person) {
    Person.name = "Jeff";
}
```

This would output;

Before swap p1: Elliott

After swap p1: Jeff

You can still change the things inside of the object, you just cant change the objects reference.

Example

```
public static void main(String[] args){
    Person p1 = new Person("Elliott");
    Person p2 = new Person("Gareth");
    System.out.println("Before swap p1: " + p1.name + " p2: " +
p2.name);
    swap(p1,p2);
    System.out.println("After swap p1: " + p1.name + " p2: " +
p2.name);
}

public static void swap(Person person1, Person person2){
    Person tempPerson = new Person("");
    tempPerson = person1;
    person1 = person2;
    person2 = tempPerson;
}
```

This would output;

Before swap p1: Elliott p2: Gareth

After swap p1: Elliott p2: Gareth

No change! Any changes to the reference that is passed to the method doesn't reflect as it's just a copy of the original reference, both point to the same place but if you change the reference of one, it won't change the reference of the original.

GARBAGE COLLECTION

Garbage collection is the act of deleting object/memory segments that are no longer in use, some languages require you do this manually however java has automated garbage collection.

Java can detect if an object no longer has any references, if it finds an object with 0 references then it will delete it.

- How can an object be unreferenced?
 - › By setting objects reference to null.
 - › By assigning the objects reference to another object
 - › By using the object **anonymously**.

```
Person p = new Person("Elliott");  
p = null;
```

```
Person p = new Person("Elliott");  
Person p2 = new Person("Gareth");  
p = p2;
```

```
new Person("Dovdoota");
```

STATIC

Static is a keyword you will have seen quite a bit before this point, and eclipse has most likely complained at you for creating things that are not static.

Static means it belongs to the class

- As opposed to the instance of the class

Say we have a Car class that has a method accelerate()

To use that, we first need to create a car object.

```
Car c = new Car();
```

And then we can call the method

```
c.accelerate();
```

However static methods don't require us to create an object, since they don't depend on an instance of the class to be used/called, so if accelerate was a static method we could do this.

```
Car.accelerate();
```

Calling the method without creating an object first.

This is good for a lot of reasons, if we want to have variables or methods that are **related** to a Car, but don't belong to a specific instance of a car, we can make them static!

The general rule of thumb is, if the method/variable makes sense without being attached/called upon the Object in question, then make it static.

Java uses a lot of static methods, such as `Integer.parseInt`, `Math.sqrt()`, since we don't need too.

CASTING

Casting is used to treat one data type as another data type.

Simple usage would be to cast a double/float into an integer, if you decide you don't need anything after the decimal point.

Casting is very important in polymorphism, if you have a hierarchy of Vehicle->Car, car **is a** vehicle.

We could create a list that contains vehicles which would accept cars and bike etc., since they inherit from vehicle.

However when we get an object out of the list, it will be treated as a vehicle object, not a car object, so any specific methods or variables that only exist in car will not be visible (although they're still inside it).

So you have to cast that vehicle to a car, to then access the car specific variables/methods.

This can be risky as the compiler will let you cast any vehicle to any subclass that inherits from it, regardless if it actually is that type, if you cast a Bike object to a Car, on runtime it will break, since a bike is not a car.

A way that we can check what an object is, is using the ***instanceof*** keyword, which compared an object to a Class, returning true or false depending on if that object is that class.

```
if(vehicle1 instanceof Car)
```

Java

Useful Eclipse Shortcuts

- **CTRL + N** – Create a new Project
- **CTRL + SHIFT + N** – Create a new file (e.g. class)
- **CTRL + Q** – Jump to last location edited
- **CTRL + .** – Jump to next syntax warning/error
- **CTRL + SHIFT + P** – Jump to matching bracket
- **ALT + UP/DOWN** – Move current line
- **CTRL + H** – Search Workspace
- **CTRL + SHIFT + O** – Import missing libraries
- **CTRL + SHIFT + F** – Format Class
- **CTRL + SHIFT + /** – Add block comment around selection
- **CTRL + SHIFT + ** – Remove block comment around selection
- **ALT + /** – Word completion
- **ALT + SHIFT + R** – Rename selected element and all references of that element
- **CTRL + F11** – Save and launch application
- **sysout + CTRL + SPACE** – Writes “System.out.println(;)”
- **Source >** Generate getters/setters/toString()/Constructors

More shortcuts:

<https://www.shortcutworld.com/en/win/Eclipse.html>

JAVA – ADDITIONAL

DESIGN PATTERNS

- Design patterns represent the best practices developed and used by experienced OO software developers.
- In general they are solutions to common problems that developers faced during development.
- Think of them as ‘Templates’ that you can use to make your code more efficient/effective.

Creational Patterns

Creational Patterns are centred around the creation of objects while hiding the logic behind the creation. The aim of this is to give the program greater flexibility in deciding which objects are needed for a given case.

Structural Patterns

Structural design patterns focus on the way the classes and objects are structured. The aim of this is to use inheritance and interfaces to define the composition of objects for new functionality.

Behavioural Patterns

Behavioural patterns are focused on the communication between objects.

Builder Pattern

- Builder pattern solves the issue where the amount of parameters that an object requires leads to an exponential amount of constructors
- (To enable every combination between the parameters)
- How the builder pattern solves this problem is by using default values and being able to return/fill a constructor at every step.
- Builder pattern is an object creation software design pattern, intention of it is to find a solution to the telescoping constructor anti-pattern which occurs when the increase of parameters for constructors leads to an exponential list of constructors.
- Instead of using numerous constructors the builder pattern uses another object, a builder that receives each parameters step by step then returns the resulting constructed object at once.

Java

Example

```
public class Trainee {
    private String name;
    private int age;
    private String technology;
    Trainee(String name, int age, String technology)
    {
        this.name = name;
        this.age = age;
        this.technology = technology;
    }
}

public class TraineeBuilder {
    private String name;
    //we dont want a default name.
    private int age = 0;
    //default age
    private String technology = "nothing"; //technology
    public TraineeBuilder() {}
    public Trainee buildTrainee()
    {
        return new Trainee(name, age, technology);
    }
    public TraineeBuilder name(String _name)
    {
        this.name = _name;
        return this;
    }
    public TraineeBuilder age(int _age)
    {
        this.age = _age;
        return this;
    }
    public TraineeBuilder technology(String _technology)
    {
        this.technology = _technology;
        return this;
    }
}

Trainee t1 = new TraineeBuilder().name("connor").
buildTrainee();
Trainee t2 = new TraineeBuilder().name("jeff").age(12).
technology("devops").buildTrainee();
```

Java

Explanation

So here we get a really tidy one-liner that basically lets you put as many or as little parameters as you want to use for that object, in the first example we just give it a name and in the second we fill everything in.

How this works is that when we create our trainee builder, after the brackets we can then call the method `name()` which returns the `TraineeBuilder`, so then on that instance of trainee builder we can call the method `age()` and so forth, until we want it to return type of `Trainee` which we then call `buildTrainee()` and that is the object that is assigned to “t2” in this case, our trainee.

There are hundreds of different design patterns that cater for all kinds of scenarios, an example list is found here:

<http://java-design-patterns.com/patterns/>

EXCEPTIONS

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

Every system is going to have errors that occur at some point, Error handling is the way in which we prevent the system from terminating unexpectedly, aka **handling the errors in a desirable way.**

There are 2 different type of exceptions

- **Checked** – An exception that is noticed by the compiler
- **Unchecked** – An exception that isn't noticed by the compiler.

CHECKED EXCEPTIONS

- Compile time
- Where we can anticipate an exception ahead of time and plan for it
- Exception is the parent class of all Checked Exceptions

Examples

- FileNotFoundException
- SocketException
- UnsupportedOperationException
- ClassNotFoundException
- NoSuchMethodException
- NoSuchFieldException

Java

UNCHECKED EXCEPTIONS

- Runtime issues (When the code is being ran)
- Caused by bad programming
- E.g. trying to address an array at a point that is larger than the arrays size
- RuntimeException is the parent class of all unchecked exceptions.
- If we are throwing a RuntimeException (or any subclass of it) in a method, it's not required to specify them in the signatures throw clause.

Examples

- `ArrayIndexOutOfBoundsException`
- `ClassCastException`
- `NullPointerException`
- `NumberFormatException`
- `IllegalArgumentException`

ERRORS

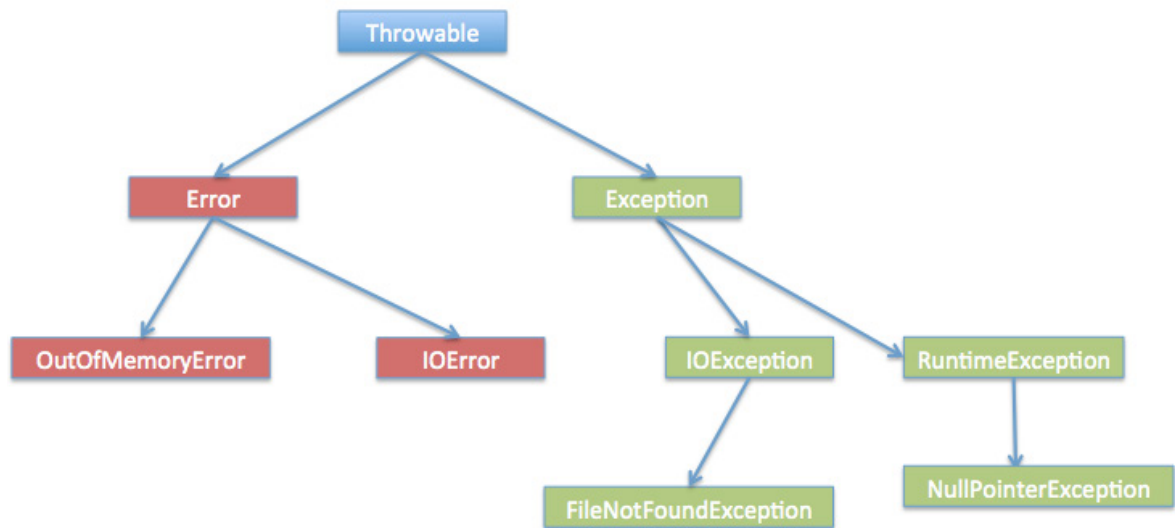
- Subclass of `Throwable`.
- Indicates a serious problem that an application should not try to catch

Examples

- `VirtualMachineError`
- `ThreadDeath`
- `OutOfMemoryError`
- `CoderMalfunctionError`

Java

EXCEPTION HIERARCHY



HANDLING EXCEPTIONS

To handle exceptions you use try & catch

You encapsulate the code you are going to try and run, in a try block.

After the try block, you try and catch an exception, then run code associated with that exception.

Syntax

```
try{
    //code
}
catch(ExceptionType exceptionVariableName) {
    //code
}
```

Java

Example

```
try{
    //io stuff
}
catch(IOException ex){
    System.out.println("Exception!!");
}
```

You can have multiple catch blocks after each other, so that they catch different exceptions and handle the exception in appropriate ways.

*Note: If you have multiple catch blocks, they must get **more** specific of an exception, so if a superclass exception is above a subclass exception, the subclass exception could never be run and will not compile.*

You can also have a finally block after the last catch statement (if any), this block will **always** be executed, either after the try block has finished or if an exception is thrown, the catch block is finished.

This is good for code that you want to run **regardless** of if an exception is thrown or not.

Example

```
try{
    //io stuff
}
catch(IOException ex){
    System.out.println("Exception!!");
}
finally{
    System.out.println("Finally!");
}
```