INFO1113 NOTES

INFO1113 Week 1 Lecture – Java Fundamentals

Java

- Java Is largely used in industry for the following reasons:
    - → OOP is the primary paradigm
    - → Flexible
    - → Portable
    - → Legacy and interoperability
    - → Static and strongly typed
    - → Open source
    - → Performant
- Java is suitable in:
    - → Servers and web backends
    - → Databases and data processing
    - → Multimedia (video games, sound generation, etc.)
    - → Networking applications
    - → Mobile applications
    - → Frontend web applications
- Where it isn't suitable:
    - → Operating systems
    - → Embedded programming

Object Oriented Programming

- Object Oriented Programming is another programming paradigm, in the same way that procedural and functional are programming paradigms.
    - → It is just a different way of writing code
    - → The idea is that objects communicate to each other
    - → Reusing and portability
    - → Encapsulation (can hide attributes)
    - → Makes your applications more robust and maintainable

Hello Java

- When learning any language, we want to be able to output simple text to the screen. So as with all programming languages we are able to write the classic “Hello World” program.

```java
public class HelloJava {

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

}
```

Compiling Programs

- Java is an interpreted and compiled language.
- Python source code is interpreted by a python interpreter and code is compiled and executed at runtime.
- C is compiled into a set of machine instructions and then can be executed by the machine.
- Java is compiled into java bytecode first and then executed by a java virtual machine.

Anatomy of a Source File

Let’s take apart a Java source file (Refer to p. 48-51, Savitch & Mock)

```java
import java.util.Scanner;

public class Anatomy {

    public static void main(String[] args) {

        System.out.println("Hello! This will output to the screen");

        int integerVar = 1;

        float f1 = 1.5;

        Scanner keyboard = new Scanner(System.in);

        String s = "This is a string!";

        s = keyboard.nextLine(); //We have reassigned it

        System.out.println(s);

    }
}
```

Class body, contains attributes and methods inside it.

Method body, scope is defined with curly braces.

Methods must be contained within a class

```java
import java.util.Scanner;

public class Anatomy {

    public static void main(String[] args) {

        System.out.println("Hello! This will output to the screen");

        int integerVar = 1;

        float f1 = 1.5;

        Scanner keyboard = new Scanner(System.in);

        String s = "This is a string!";

        s = keyboard.nextLine(); //We have reassigned it

        System.out.println(s);

    }

}
```

Opening brace, creates a scope which variables and instructions are executed.

Closing brace, shows where the scope ends.

(another example, Listing 1.1 on p. 48, Savitch & Mock)

Let's take apart a Java source file (Refer to p. 48-51, Savitch & Mock)

```java
import java.util.Scanner;

public class Anatomy {

    public static void main(String[] args) {

        System.out.println("Hello! This will output to the screen");

        int integerVar = 1;

        float f1 = 1.5;

        Scanner keyboard = new Scanner(System.in);

        String s = "This is a string!";

        s = keyboard.nextLine(); //We have reassigned it

        System.out.println(s);

    }

}
```

Access modifier, allows the following definition to be accessible by others

Class keyword, used when defining a class.

Class name must go next.

Class name, since the public keyword was used, the filename must be

Anatomy.java

Let's take apart a Java source file (Refer to p. 48-51, Savitch & Mock)

```
import java.util.Scanner;

public class Anatomy {

    public static void main(String[] args) {

        System.out.println("Hello! This will output to the screen");

        int integerVar = 1;

        float f1 = 1.5;

        Scanner keyboard = new Scanner(System.in);

        String s = "This is a string!";

        s = keyboard.nextLine(); //We have reassigned it

        System.out.println(s);

    }

}
```

Static modifier allows it to be accessed without the need of an object.

main method, this is the first function invoked in your java program (starting point)

Command line arguments. It is of type String and [] defines it as an Array.

Let's take apart a Java source file (Refer to p. 48-51, Savitch & Mock)

```
import java.util.Scanner;

public class Anatomy {

    public static void main(String[] args) {

        System.out.println("Hello! This will output to the screen");

        int integerVar = 1;

        float f1 = 1.5;

        Scanner keyboard = new Scanner(System.in);

        String s = "This is a string!";

        s = keyboard.nextLine(); //We have reassigned it

        System.out.println(s);

    }

}
```

Outputting to the screen, we line the string.

String literals are defined using double quotes. E.g

"This is a String"

Primitive type int which is a 32bit signed integer followed by a variable name and initialised to 1.

float type with a different label name.

Reference type String variable set to string literal "This is a string!"

(For another example, Listing 1.1 on p. 48, Savitch & Mock)

Let's take apart a Java source file (Refer to p. 48-51, Savitch & Mock)

```
import java.util.Scanner;

public class Anatomy {
    public static void main(String[] args) {
        System.out.println("Hello! This will output to the screen");
        int integerVar = 1;
        float f1 = 1.5;
        Scanner keyboard = new Scanner(System.in);
        String s = "This is a string!";
        s = keyboard.nextLine();//We have reassigned it
        System.out.println(s);
    }
}
```

Imports the **Scanner class** so that it can be used within your **class**

Creates a scanner **object** which allows you read input from standard input (**stdin**)

Let's take apart a Java source file (Refer to p. 48-51, Savitch & Mock)

```
import java.util.Scanner;

public class Anatomy {
    public static void main(String[] args) {
        System.out.println("Hello! This will output to the screen");
        int integerVar = 1;
        float f1 = 1.5;
        Scanner keyboard = new Scanner(System.in);
        String s = "This is a string!";
        s = keyboard.nextLine();//We have reassigned it
        System.out.println(s);
    }
}
```

Assigns s to the next line of input that user has given.

Outputting the string using **println**.

This implicitly calls **.toString()** on reference types or uses an **overloaded println** method

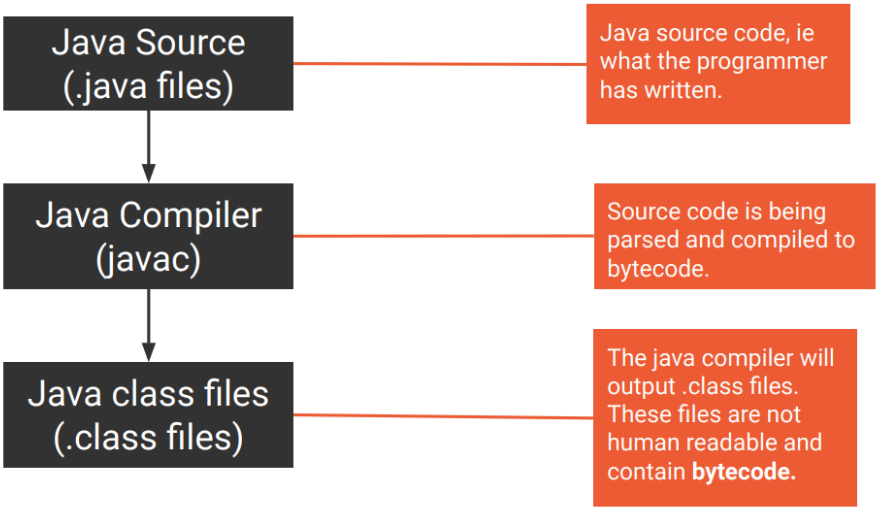
or another example, Listing 1.1 on p. 48, Savitch & Mock)

Compilation

- Java is a compiled language and therefore requires another step before executing any code → compilation step will transform the source code to bytecode.

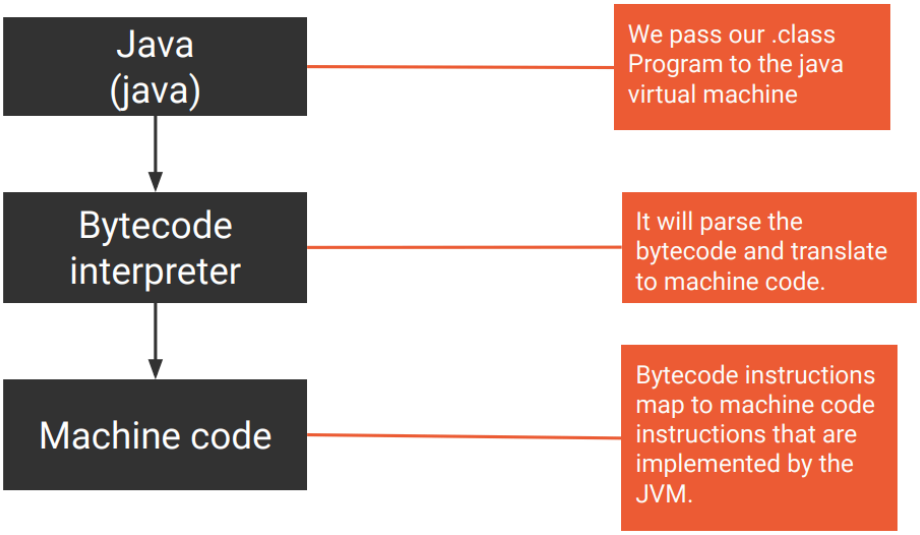
```
> javac MyProgram.java
> java MyProgram
Hello Everyone! This is a java program
```

```
> javac MyProgram.java
```



After this process has been successfully completed we are able to now execute our program using the **java virtual machine (jvm)**

```
> ls
MyProgram.java    MyProgram.class
> java MyProgram
```



During this process our program is executing and is contained within the **java virtual machine**.

Scoping

- A block is a collection of statements that is delimited by a pair of curly braces {}.

```
public class Scoping {

    public static void main(String[] args) {

        int x = 0; //x starts here
        {

            int y = 1; //y starts heres
            {
                int z = 2;
            }// z ends here
        } // y ends here

    } //x ends here

}
```

Types and Variables

- Unlike interpreted languages like python, ruby or JavaScript types, Java has a separation of primitive types and reference types.
- String is a reference as it is an aggregation of the char type (array of char elements).
- Any class you create is a reference type → we are working with a binding to a memory address, not the object itself.

Primitive Types

Name	Kind	Memory	Range	Type
boolean	Boolean	1 byte	true or false	**primitive**
byte	integer	1 byte	[-128, 127]	**primitive**
short	integer	2 bytes	[-32768, 32767]	**primitive**
int	integer	4 bytes	[-2147483638, -2147483637]	**primitive**
long	integer	8 bytes	[~-9 × 10^{18}, ~9×10^{18}]	**primitive**
float	floating-point	4 bytes	[±3.4× 10^{38}, 1.4×10^{-45}]	**primitive**
double	floating-point	8 bytes	[±1.8× 10^{308}, ±4.9×10^{-324}]	**primitive**
char	character	2 bytes	[0, 65535]	**primitive**
String	string	variable	[0, very long]	**object**

Why Do Types Exist?

- We are able to confirm the type that is being assigned. The compiler can check the assignment of variables (in the event we are attempting to assign float to an int or some other nasty assignment).
- Clear allocation of memory. With type information, the compiler knows how much memory should be allocated for that variable.
- The type systems helps prevent type errors from occurring, This allows the compiler to verify interaction of variables and memory boundaries.

Types and Variables

What would be the output of:

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

    int i = 1;
    double f = 1.0;

    System.out.println(i/2);
    System.out.println(f/2);
    System.out.println(f/i);

}
```

Two integers are involved and this is where integer division occurs. Since i is assigned to 1, the divison will be 0 (not 0.5 as .5 cannot be represented as an integer).

Since a floating point number is involved during the calculation it will promote the integer (2) to a float

What is the result of this operation?

Command Line Arguments

- Java inherits C like command line arguments with few differences:
  - → the program name is not included in the arguments
  - → Java has a String type while C does not

```
public class CommandLineArgs {

    public static void main(String[] args) {

        String arg1 = args[0];

        System.out.println(arg1+"!");
    }

}
```

```
> java CommandLineArgs Hi
Hi!
```

Arrays in java start at index 0.

We have assigned String arg1 to the first element in the args array

Scanner and Input

- Scanner is an abstracted object that provides an interface for reading data.

```
import java.util.Scanner;

public class UsingScanner {

    public static void main(String[] args) {

        Scanner keyboard = new Scanner(System.in);

        //We want to read in a line from Standard Input

        String s = keyboard.nextLine();

    }
}
```

Scanner object declared and initialised and we read the next line inputted by the user using the nextLine() method.

Boolean

- Java inherits logical (and other operators) from C.
- We have the same logical operators for and, or and not.

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

    boolean t = true;
    boolean f = false;

    boolean exampleAnd = t && t; //This is true
    boolean exampleOr = f || t; //This is true
    boolean exampleNot = !t; //This is false

}
```

logical **and**. It will check if **both** of the expression are true.

logical **or**. It will check if **one** of the expression of either side is true.

This will enact the inverse of the expression. If **t** is **true**, **!t** is **false**

If Statements

- Similar to other languages, Java also has if statements. However, they have a strict requirement that that statement is of type Boolean.

```
public class IfProgram {

    public static void main(String[] args) {

        int t = 1;
        int f = 0;

        if(t) {
            System.out.println("This will be executed");
        }

        if(f) {
            System.out.println("This will not be executed");
        }
    }
}
```

In other languages true and false are typically defined as 1 and 0 respectively.

However java enforces the **correct type** to be used

We would encounter the following error:
| Error:
| incompatible types: int cannot be converted to boolean

INFO1113 Week 2 Lecture – Control Flow, Loops, Static Methods and Contiguous Memory

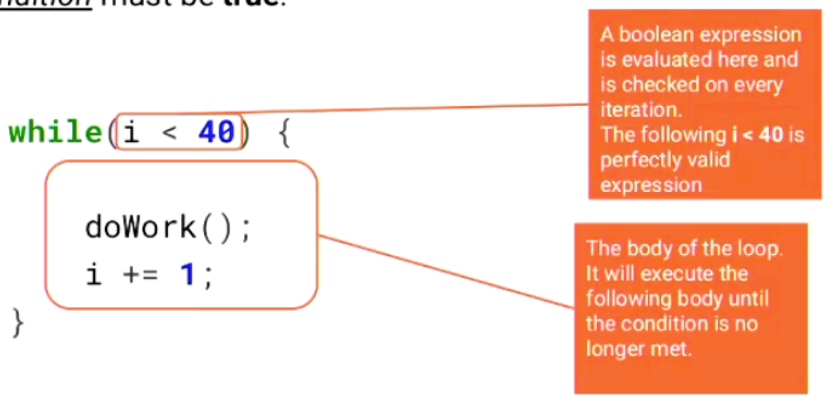
Loops

- There are four types of loops we can write within Java.
  - → while
  - → do-while
  - → for
  - → for-each
- The constructs are part of the language's syntax and typically follow a similar pattern.

While Loop

Syntax: `while (condition) statement`

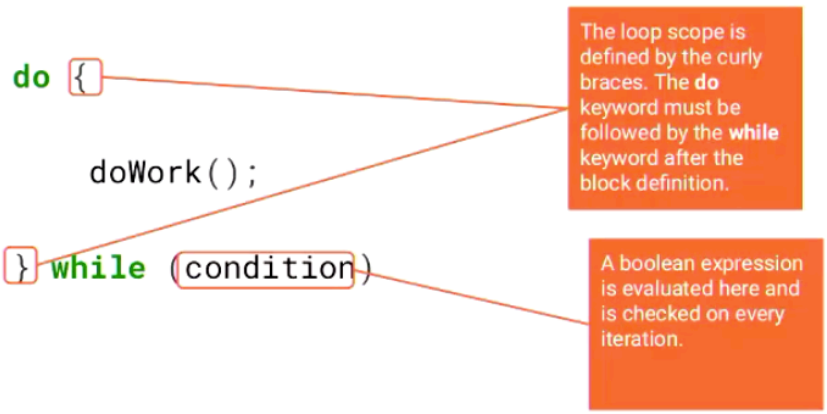
As with if statements, for this branch to start and *continue* execution the *condition* must be **true**.



Do While Loop

Syntax: `do {} while(condition) statement`

Similar to the while loop but it will always execute the block at-least ***once*** and *continue* execution if *condition* is **true**.



- Do-While is known to be discouraged in many style guides.

For Loop

- For loops can always be rewritten as a while loop.

Syntax: `for( [variable]; [condition]; [update])` statement

`for` loops are broken up into 3 separate sections. **Variables, Conditions** and **Updates** sections.

```
for( [variable]; [condition]; [update] )
{

  doWork();

}
```

- Variable: created and initialised in the first argument (often a counter) → they will be restricted to the loop's scope.
- Condition: Boolean expression inputted here → no different to a while loop.
- Update: update any variables defined within the variable section (or variables defined in the outer scope).

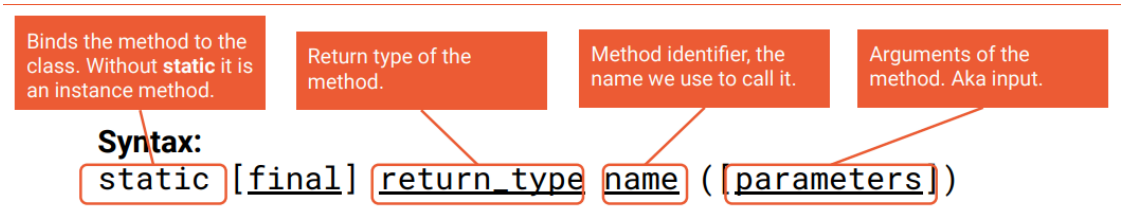
For-Each Loops

Syntax: `for( binding : collection )` statement

`for`-each loops involve the use of **iterators** (exception being arrays).

- Anything kind of object that implements hasNext(), next(), and optionally remove() aggregates other objects.
- An array index is missing by using a for-each loop.

Static Methods



A method is a stored set of instructions bound to an object. In the case of a static method, the object is the class which it is defined in.

Example:

```
public static int addThree(int a, int b, int c)
{
    return a+b+c;
}
```

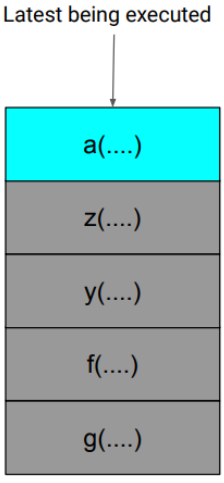
- The return type can be any primitive type, reference type or void.

Return Types

- Java can use any primitive or reference type as a return type. The compiler will check and ensure that any assignment to the return value of a method is correct.
- There is a special return type called void that does not return any value and any void method is typically used for manipulating passed data or output.
- Return data from a method is generally used for querying or object creation.

Call Stack

- Java is a stack-based language so when a method is executed, it is put onto a call-stack.
- The method being executed at the top of the stack is the most recently called method.
- A method finishes executing once it has reached a return state or for a void method, once it has reached the end of the method scope.
- Each method executed gets a Frame allocated and a frame will hold data, partial results, return values and dynamic linking.
- A Frame is created when a method is invoked at runtime by the Java virtual machine.



Arrays and Initialisation

- An array is a contiguous block of memory containing multiple values of the same type.
- When an array is initialised the array will be allocated and will return the address of where the array is stored.
- We are also able to solve the problem of having many variables.

```
variable01 = 32;
variable02 = 98;
variable03 = 34;
...
variable98 = 23;
```

- We can access the 98th element if we have an array of size 100.

Initialisation

- Within the java language, we are able to initialise an array in a few different ways. Most commonly and what is typically used is to allocate and specify size.

```
int[] numbers = new int[16];
```

- However, we can always initialise using static initialisation.

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

- This translates to an array of length 4, containing the elements 1, 2, 3, 4. Similarly, we can initialise an array like so.

```
int[] numbers = new int[] {1, 2, 3, 4};
```

- This is more commonly used when passing an array with values known at compile time. This is because the compiler knows the type being passed and what values it should contain.

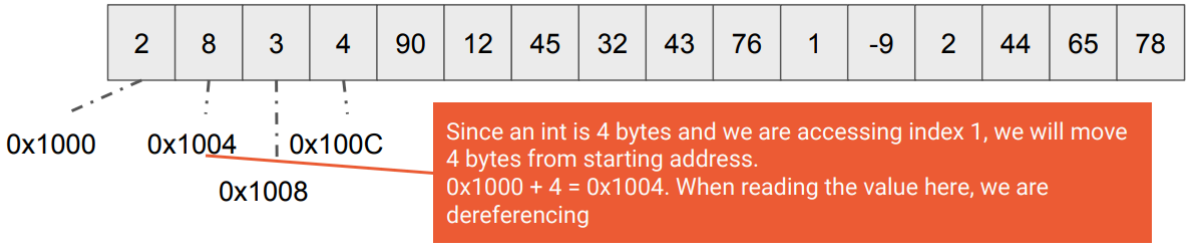
Array Layout

- Considering the layout, since we know that it returns the starting point of the array, how does it navigate the array.

Let's say we have the current array:

```
int[] numbers = new int[16];
```

And contains the following numbers:



- Knowing the type and the size and using the new operator, the compiler derives the idea that we want to dynamically allocate 16 integers.

Reference and Primitive Type Arrays

- Primitive type arrays and reference type arrays differ in the sense that reference types initialise 4 references, which in this instance is a memory address.
- This is infers the reference type arrays do not contain a string but a reference to a string.

```
String[] numbers = new String[4];
```

General Rules with Array Initialisation

- Primitive integer types (byte, short, int, long) are initialised to 0 by default.
- Default value of elements of a Boolean array are false.
- Floating point numbers such as float and double are initialised to 0.0f and 0.0d respectively.
- Elements of a char array are initialised to \u0000.

```
{ '\000', '\000', '\000', '\000' }
```

- Any Reference type is initialised to null.

```
{ null, null, null, null }
```

Multi-Dimensional Arrays

- We are able to create multi-dimensional arrays of two types → one adheres to a matrix-like structure and the other is commonly referred to as a jagged array.

```
int[][] array = new int[3][3];
```

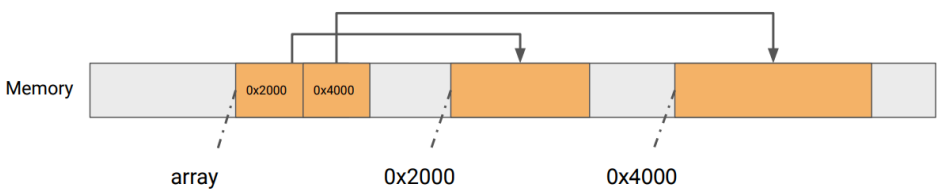
```
int[][] array = new int[3][];
```

- Arrays are also reference types. When initialised, the variable array will contain 3 null elements. We are able to specify lengths on each element.

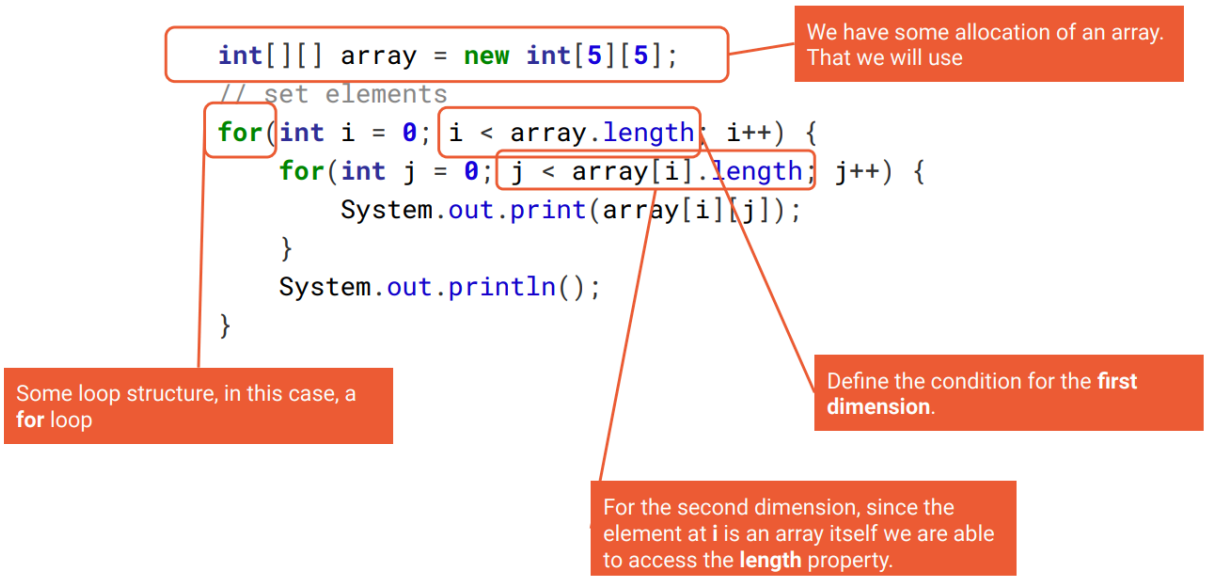
```
array[0] = new int[5];
array[1] = new int[10];
```

- With the following code, it will initialise the 2 elements within the array to null. This allows us to set each element to a separate array.

```
int[][] array = new int[2][];

array[0] = new int[4];
array[1] = new int[8];
```



- We can traverse a multidimensional array like below:



- We can output the element at [i][j] → assuming we set all the elements in the array from 0-24:

```
> java ArrayOutput
0 1 2 3 4
5 6 7 8 9
10 11 12 13 14
15 16 17 18 19
20 21 22 23 24
<program end>
```

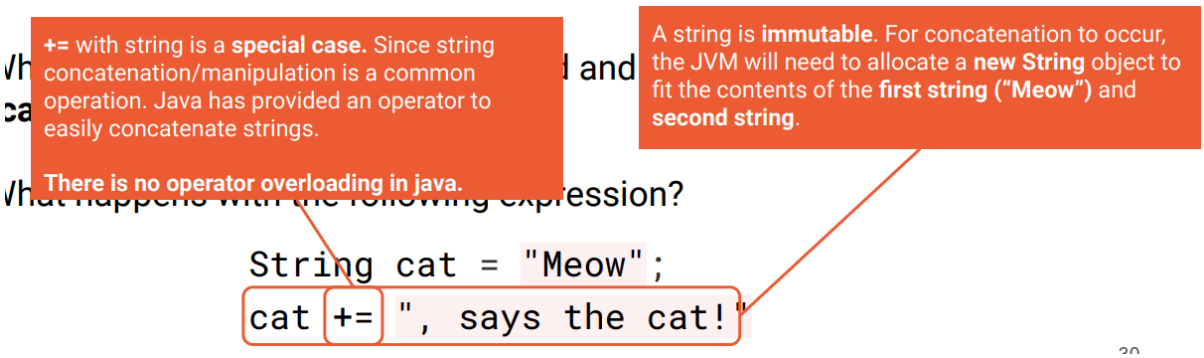
- However, it depends on your use case. There are many different ways to traverse an array and we have to keep in mind the way it needs to be accessed e.g.
    - → image resizing
    - → blur filters
    - → edge detection
    - → animating entities
    - → near pair of points
- All have different schemes and you will need to keep in mind how a domain specific formula dictates accesses.

Strings

- String is a reference type that aggregates characters. However, like many other programming languages, Java treats Strings as immutable.
- When initialising a string type, the JVM will allocate memory to contain the string.

```
String cat = "Meow";
```

- When assigned, the string is allocated and binded to the variable “cat”.



- The same operation will occur if we were to reassign with the equivalent expression.

Comparing Strings

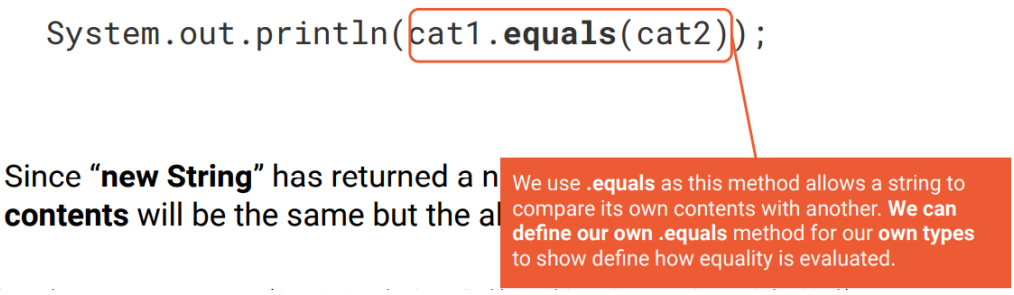
- Any reference type variable holds onto the memory reference of the object.
- With the following code, the output will be true.

```
String cat1 = "Meow";
String cat2 = "Meow";

System.out.println(cat1 == cat2);
```

- With the code below, the output will be false.

```
String cat1 = "Meow"; -> 0x1000
String cat2 = new String("Meow"); -> 0x2000

System.out.println(cat1 == cat2);
```

- Reference type does not implicitly have the ability to compare itself to another type (besides reference) without first defining a method.
- To compare them, we need to write a method to compare them → using the == operator is testing the equivalence of the memory reference.



Beware the String Pool and Equality

- When writing our first program, we have been able to define a string literal. However, java employs an intelligent but often mistaken optimisation for strings.

```
String cat1 = "Meow"; -> 0x1000
String cat2 = "Meow"; -> 0x1000
```

- If a string literal is specified, it will be added to a string pool → this allows the compiler to optimise for memory storage.
- The compiler will use the same allocation and provide the same reference to a string variable that refers to the same literal.

Mutating String (Welcome StringBuilder)

- Recreating a new string and deallocating the old one can be costly for the JVM.
- We are able to mitigate this by using a class called StringBuilder → contains an internal array of characters and is mutable but does not have the same kind of affordances as String.

```
StringBuilder b = new StringBuilder();
```

- The StringBuilder class allows us to assemble a string and resize the internal character array when the number of characters exceeds the capacity of the array.

Why Have Both Types?

- Each time we use += with the String type, we are creating a new string → the String class is immutable, and this can have great benefits for ensuring we have a read only or for simple concatenations.
- However, when it comes to complicated string manipulation or excessive string manipulation, we will need to use the StringBuilder.

```
StringBuilder b = new StringBuilder();
b.append("Hello");
b.append(" World");
```

INFO1113 Week 3 Lecture – Objects, Classes, UML, Methods and IO

Classes

- The clear distinction between a primitive type and a reference type is that reference types ARE classes.
- "A class defines a type or kind of object. It is a blueprint for defining the objects. All objects of the same class have the same kinds of data and the same behaviours. When the program is run, each object can act alone or interact with other objects to accomplish the program's purpose."
- It is simply conveyed as a blueprint/template/concept of an object → the primary way of structuring data in Java.

Objects

- The type of an object variable is its class.

```
Point p;
```

Shiny **new** keyword! As discussed last week, this allocates memory and instantiates an object.

Objects are *instance* of a particular *class*.

```
Point topleft = new Point(-1, -1);
Point right = new Point(1, 0);
Point home = new Point(-3388797, 15119390);
```

We have to ask what where this **method** exists

Creating Classes

- We can start with a basic class definition, and then instantiate it with a code like the below.

```
public class Cupcake {
    boolean delicious;
    String name;
}
```

This is the **body** of the class. We define **attributes** of **object** within this space.

We can **instantiate** this class with the following line of code

```
Cupcake c = new Cupcake()
```

Declared a **Cupcake** object.

Java is **allocating** space for a **Cupcake** object and **invoking the constructor** to initialise it.

Constructors

- Every class in Java has a Constructor even if it isn't explicitly defined → its role is to construct an object, provide default values etc.

Extending our **Cupcake** class we can write our own constructor.

```
public class Cupcake {
    public boolean delicious;
    public String name;

    public Cupcake(boolean isTasty) {
        delicious = isTasty; //Aren't they all?
        name = "Chocolate Cupcake";
    }
}
```

We can expand on this to provide **default values** and **parameters** for our constructor . We can then invoke the parameter with arguments that relate to the object.

Now using our nice cupcake class, let's see what we can do with it!

```
public class Cupcake {
    public boolean delicious;
    public String name;

    public Cupcake(boolean isTasty) {
        delicious = isTasty;
        name = "Chocolate Cupcake";
    }
}
```

We can access the attributes of the object by using the **.<attribute>**

Let's instantiate our own instance!

```
Cupcake mine = new Cupcake(true);
Cupcake toShare = new Cupcake(false);
System.out.println(mine.delicious);
System.out.println(toShare.delicious);
```

You may have already of picked up on that from using **Scanner** and **String**

Instance Methods

- An instance method operates on attributes associated with the instance. These methods can only be used with an object.

Syntax:

```
[final] return_type name ([parameters])
```

Let's extend our **Cupcake** class!

```
public class Cupcake {
    public boolean delicious;
    private String name;

    public Cupcake(boolean isTasty, String cupcakeName) {
        delicious = isTasty;
        name = cupcakeName;
    }

    public void setName(String n) { name = n; }

    public String getName() { return name; }
```

A **getter** method has been specified here. This method merely returns the attribute **name.**

private modifier limits how where the attribute can be accessed. **public** allows access outside of the class while **private** limits itself to the scope of the class.

A **setter** method specified. This allows us to modify the **name** attribute

```
Cupcake mine = new Cupcake(true, "My Cupcake!");
Cupcake toShare = new Cupcake(false, "Everyone's Cupcake");
mine.setName("My Cupcake, Don't touch!");
```

If we were to use **.getName()** on **mine** and **toShare**. What would the return value be?

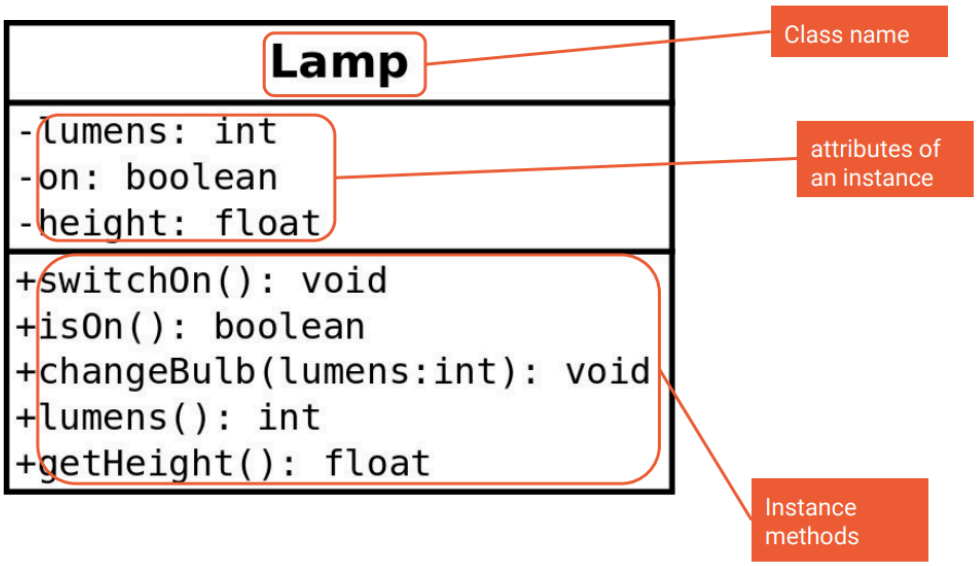
```java
public class Cupcake {
    public boolean delicious;
    private String name;
    public boolean eaten;

    public Cupcake(boolean isTasty, String cupcakeName) {
        delicious = isTasty;
        name = cupcakeName;
        eaten = false;
    }

    public void setName(String n) { name = n; }

    public String getName() { return name; }

    public void eat() {
        if(!eaten) {
            System.out.println("That was nice!");
        eaten = true;
    }
}
```

Expanding on this method, we can output to the user when it has been eaten.

38

UML

- The Unified Modelling Language is a visual language to assist with designing applications and systems.
- UML offers the ability to purely design a system in how objects will interact with each other as well as describing interactions a user may have with the system.
- Class diagrams allow us to design classes before implementing them, giving the ability to model the system without implementing it first.

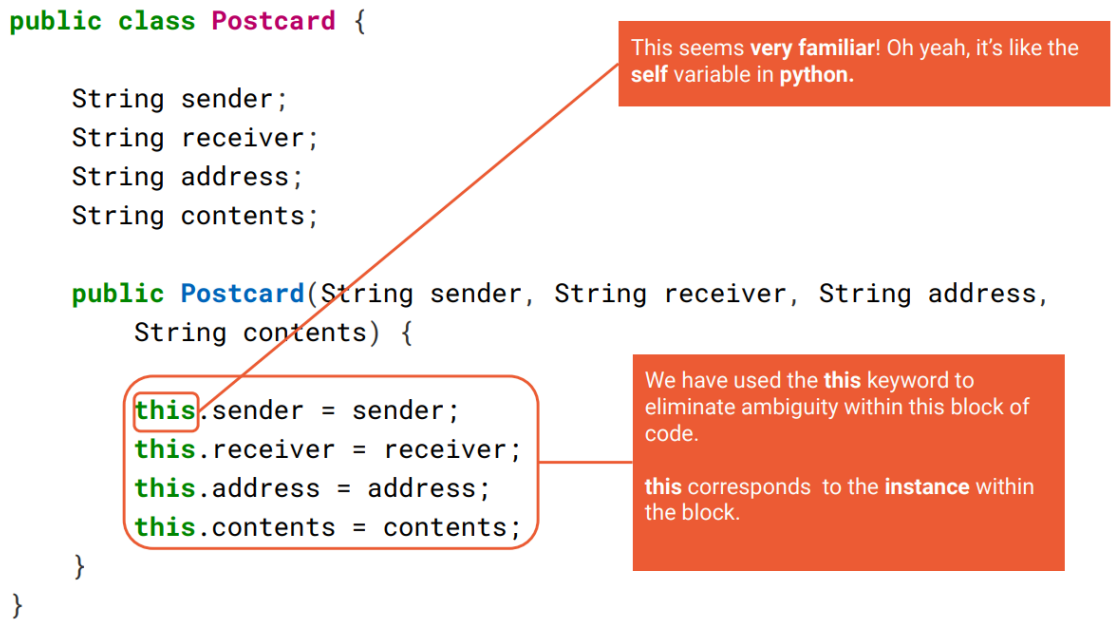


- The – annotations specifies private, whilst + specifies public.

this Keyword

- The this keyword can help with eliminating ambiguity and is also used for passing an object reference within an instance context.

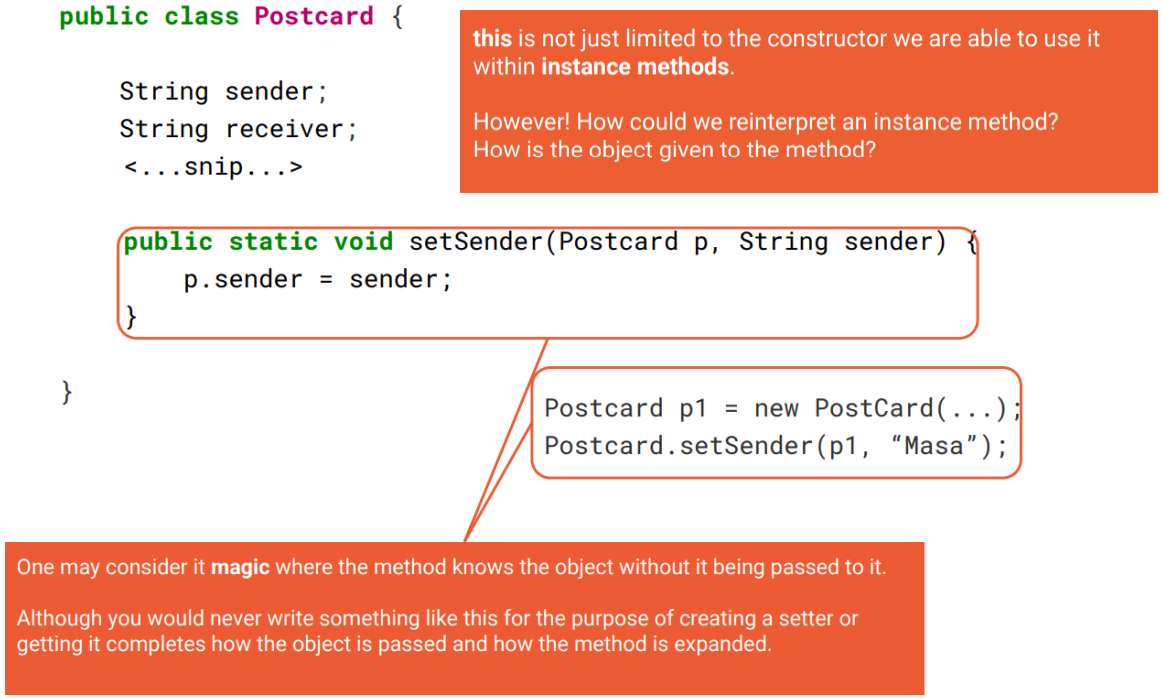
- The this keyword allows the programmer to refer to the object while within an instance method context. We cannot use the keyword within a static context.
- It is also used for referring to another constructor to allow for code reusability.



- If the additional line System.out.println(this) was added, unique addresses will be printed e.g. Postcard@3cd1a2f1.

Expanding and Reinterpreting on Instance Methods

- Within the context of an instance method, it refers to the current calling object → it cannot be used within a static method as it is unable to refer to the calling object.



Let's examine the following code segment.

```
public class Postcard {

    String sender;
    String receiver;
    boolean received;
    <...snip...>

    public static boolean inTransit() {
        return  !received;
    }

    public void setSender(String sender) {
        this.sender = sender;
    }

}
```

This **static** method is attempting to utilise an **instance** variable. Why is this a problem?

Because it isn't referring to an object
Instance methods are not allowed in this context.

- For the setSender() method however, there is an object instantiated and we are able to use this method.

```
public class Postcard {

    String sender;
    String receiver;
    boolean received;
    <...snip...>

    public boolean alreadyArrived() {
        return hasArrived(this);
    }

    <...snip...>
    public static boolean hasArrived(Postcard p) {
        if(!p.inTransit()) { return true; }
        else { return false; }
    }
}
```

We have an **instance** method invoking a **static method** while also using the **this** keyword.

Yes! We are just passing the instance to a static function. This is no different from what we have done before but we are using the **this** keyword.

26

Input and Output

- We are able to read and write to devices. Specifically, we will be focussing on reading and writing to storage.
- If we are intending to use data stored in a file, we have to understand how that data is stored and what will be an appropriate tool for the job.

Files

- I/O Classes → within the java api, we have access to a large range of I/O classes.

Using Scanner

- We can use Scanner to read files. As the name implies, it scans for input and provides functionality to read it.

```
import java.io.File;
import java.util.Scanner;
public class FileHandle {

    public static void main(String[] args) {
        File f = new File("README");
        Scanner scan = new Scanner(f);

    }
}
```

We have **File** object that will abstract represent the the file stored at a **Path**.

Scanner accepts a file as an argument and is able read contents there.

- This will NOT compile:

```
public static void main(String[] args) {
    File f = new File("README");
    try {
        Scanner scan = new Scanner(f);
    } catch (FileNotFoundException e) {
        System.err.println("File not found!");
    }
}
}
```

Java forces us to provide some checks to ensure we are handling certain except cases correctly.

- If the file doesn't exist, we are unable to read from it. This allows the programmer to have a branch for both. A state where we can read data and one without reading data.

How is Reading Performed?

- Reading any kind of file is analogous to working with contiguous memory.
- Let's say we have the following file called 'today.txt' which contains the following elements:

  → 1. Today is great!

- Scanner itself doesn't support reading character by character because the idea of a character depends on how it is encoded.

This can be represented with the following array:

T	o	d	a	y		i	s		g	r	e	a	t	!	\0

```
File f = new File("README");
Scanner scan = new Scanner(f);
scan.next(); //Today
scan.next(); //is
scan.next(); //great!
```

Executing the following line will move the cursor to the next space (or whatever token we want to separate words by).

```
> javac FileHandle.java

FileHandle.java:7: error: unreported exception FileNotFoundException;
Must be caught or declared to be thrown
     Scanner scan = new Scanner(f);

1 error
```

As with most **IO operations** we will be required to perform some exception handling.

Writing Text Data

- Scanner only performs reading an object.
- PrintWriter allows for printing formatted representations of objects to a text-output stream.

```
import java.io.File;
import java.io.PrintWriter;
import java.io.FileNotFoundException;
public class FileHandle {
    public static void main(String[] args) {
        File f = new File("README");
        try {
            PrintWriter writer = new PrintWriter(f);
            writer.println(1.0);
            writer.println(120);
            writer.println("My String!");
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

This will write output 1.0, 120 and "My String!" to the file **README**.

We have a class that allows for writing of formatted data. It's methods are very similar to that of **System.out.** That is no coincidence!

INFO1113 Week 4 Lecture – Binary IO, Memory and Collections

Binary IO

- Like text data, we can employ patterns to read binary files and store data in a non-readable format, but they are not easily obvious to inspect.
- When reading a binary file, there is a specific layout to interpret it correctly. This is typically bundled with a file format specification.
- For example, a poke.dex file format.

pokedex_header: 40 bytes	seen_pokemon: seen*4 bytes
magic: 4 bytes : int owner: 32 bytes : char no_pkmn: 2 bytes : short seen: 2 bytes : short	pkmn_id: 2 bytes : short type_1 : 1 byte : byte type_2 : 1 byte : byte

- We also need to know about the endianness of the data as well → without this information, we may not interpret the data correctly and therefore get strange values.
- In the examples below, we will assume machine default.

```
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException

public class BinaryReader {

    public static void main(String[] args) {
        try {
            FileInputStream f = new FileInputStream("some_file.bin")
            byte[] buffer = new byte[4];
            f.read(buffer);
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

We need to **interpret** the data.

Converting Data

```
import java.io.FileInputStream;

public class BinaryReader {

    public static int convert(byte[] b) {
        return (b[3] & 0xFF) |
          ((b[2] & 0xFF) << 8) |
          ((b[1] & 0xFF) << 16) |
          ((b[0] & 0xFF) << 24);
    }
}
```

Considering we have captured the data in a **byte[]** array

We need to transform this to an integer.

Shifting and Bitwise Operations

```
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;

public class BinaryWriter {
    //<snip> Assume we have convert here
    public static void main(String[] args) {
        try {
             FileInputStream f = new FileInputStream("some_file.bin")
             byte[] buffer = new byte[4];
             f.read(buffer);
             int v = convert(buffer);
             System.out.println(v); //Output the integer
        } catch (FileNotFoundException e) {
             e.printStackTrace();
        } catch (IOException e) {
             e.printStackTrace();
        }
    }
}
```

Writing

```
import java.io.FileOutputStream;
import java.io.FileNotFoundException;
import java.io.IOException;

public class BinaryWriter {
    //<snip> Assume we have convert here
    public static void main(String[] args) {
        try {
             FileOutputStream f = new FileOutputStream("some_file.bin")
             int v = 50;
             byte[] buffer = convert(v);
             f.write(buffer); //Write out the bytes
        } catch (FileNotFoundException e) {
             e.printStackTrace();
        } catch (IOException e) {
             e.printStackTrace();
        }
    }
}
```

We write the byte[] array to the file.

Let's write some files!

```
import java.io.FileInputStream;

public class BinaryReader {

    public static byte[] convert(int v) {
        byte[] b = new byte[4];
        b[0] = (byte) (v >> 24);
        b[1] = (byte) (v >> 16);
        b[2] = (byte) (v >> 8);
        b[3] = (byte) v;
        return b;
    }
}
```

From the integer we need to transform this to a byte array for writing.

How are Classes Encoded?

- .class files are not human readable, just like image formats and executables but how does the JVM understand interpret this file?
- Simply, there is some method of interpreting binary files and being able to read them. By following the specification on Oracle[1], we can design a program to interpret and understand the class.
- However, for demonstration purposes, let's start off with something that is a little more digestible.

INSERT PHOTO

Is There a Better Way?

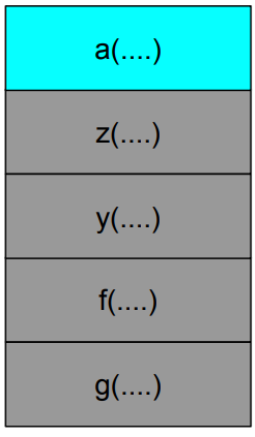
- Of course! We just showed the hard way of reading and writing, We can do away with these conversions entirely using ByteBuffer and ByteOrder classes to manipulate the data with its inbuilt methods.
- We can even go a step further and use a DataInputStream and DataOutputStream and use methods such as writeInt and readChar.

Applications

- There are lots of applications where the data itself will not be presented in a textual representation.
    - → interacting with devices and peripherals (gamepads, midi devices)
    - → modifying executables
    - → encoding images
    - → writing your own file format for your programs
    - → writing software for automotive applications
    - → video and audio streams
    - → networking application

Stack

- Every time we invoke a method, it will be placed on the stack.
- We need to understand the fixed limit of the stack within the JVM. By default, the stack limit is 1MB.
- A stack frame can be thought of as an instance of a method. Each method has a size which is the combination of its instructions, variables and argument data.
- Each stack frame has a return address.



Heap

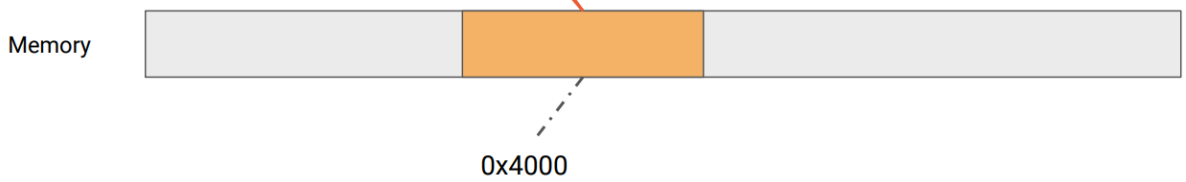
- The heap is separate memory space for which objects are dynamically allocated.
- Whenever we use the new keyword, memory for the object is allocated and an address is returned.
- Unlike the stack, heap allocated object lifetimes are a little more complicated.
- Stack based objects exist in lexicographical scope (i.e. the {} braces) while heap objects can be aliased and their lifetimes are dictated when all references are no longer in scope.

```
int[] array = new int[16];
```

Let's consider the following

```
public static void main(String[] args) {
  int x = 5;
  int[] array = new int[16];
  int[] alias = array;
  int y = x;
}
```

We can see that **array** has its own memory space which of the size of 16 integers. What would occur in the following assignment with **alias**.

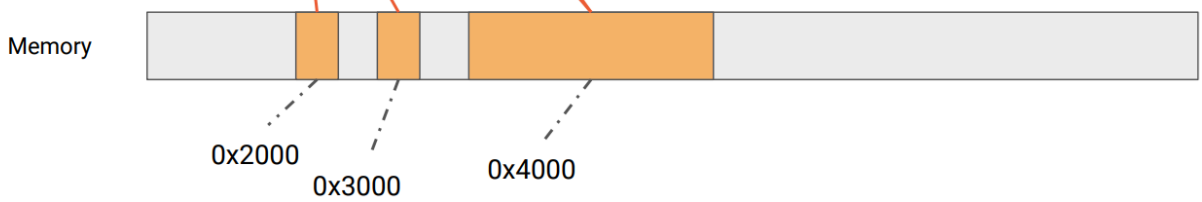


31

Let's consider the following

```
public static void main(String[] args) {
  int x = 5;
  int[] array = new int[16];
  int[] alias = array;
  int y = x;
}
```

Although, **x** and **y are on the stack** we will compare the **copy-semantics** between a **primitive type** and **a reference type.** Primitive types will copy the value between instead of the address.



33

Garbage Collector

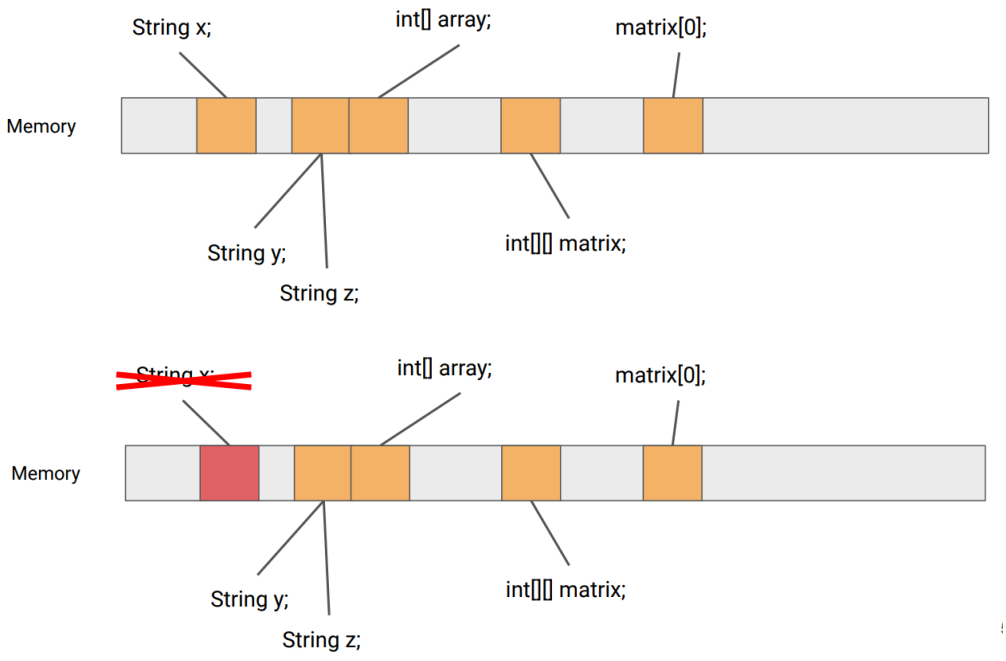
- Java employs the use of a garbage collector to free memory. Any memory that is allocated using the new keyword will be kept on the heap.
- The garbage collector will subsequently free any allocation that no longer has a reference during execution.
- The garbage collector will stop-the-world and act on allocations without a reference.

```
Person createPerson() {
  Person p = new Person();
  //things
  return p;
}

//<other bits of code>

void deletePerson(Person p) {
  delete p;
}
```

Once the allocation has been used and the programmer wants it deleted it is sent to the **deletePerson** method

- When an allocation no longer has a reference to it, the garbage collector will mark it out for deletion. This is when all references have gone out of scope.

Let's see how the following works:



etc.

Scenario 1: Forgot to Free

When we forget to deallocate:

```
Person createPerson() {
  Person p = new Person();
  return p;
}

//<other bits of code>

void main(String[] args) {
  Person p = createPerson();
  someWork(p);
  p = createPerson();

}
```

Since the original allocation has no references to it, when the GC passes it, it will be marked for deletion.

Scenario 2: Thought It Was Still Allocated

```
Person createPerson() {
  Person p = new Person();
  return p;
}

//<other bits of code>

void main(String[] args) {
  Person p = createPerson();
  Person c = p;
  delete p;
  someWork(c);

}
```

Uhoh! If the memory has been freed then potentially we could be using memory that we don't own anymore

We don't do this anymore

```
void main(String[] args) {
  Person p = createPerson();
  Person c = p;
  delete p;
  someWork(c);

}
```

Simply: There is no manual deallocation

Why?

- It's about simplifying writing software for everyone. It allows you as a programmer to be able to write software without needing to worry about memory management.
- We can minimise errors and make programming safer through this method.

Abstract Data Types

- An abstract data type is a type that is an aggregation of other types. However, we may have an ADT that only has methods and no data.
- Primitive types is not an ADT, reference types is an ADT.
- The language does not afford us to use primitive types in an abstract way → we need to do the hard work for it.

Collections

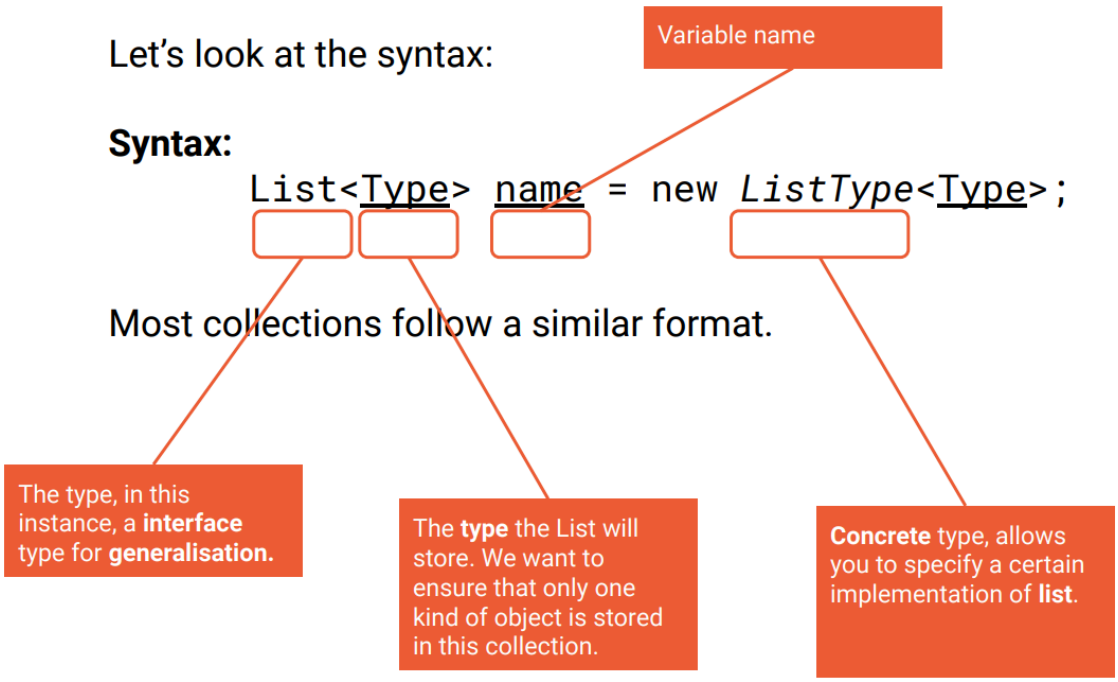
- Our applications will require certain needs in storing and organising data. Collections or aggregate types allow us to store data in the appropriate objects.
- For example:
    - → A school contains students.
    - → A stage has performers.
    - → Running times from athletes for a race.
    - → TV shows on Netflix.
- We will be visiting:
    - → List types
    - → Set types
    - → Map types

What is a List?

- The most common list type that is used is an ArrayList. This comes from the convenience of having a resizable Array.
- The other list types are:
    - → LinkedList
    - → Vector
    - → Stack
- However, depending on how we store, update and access the data will determine what type we want to use.
- "We want to store all input by the user in a collection and be able to review it" → HOWEVER, arrays are fixed length.

ArrayList (Dynamic Array)

- Not to be confused with dynamically allocated array. The ArrayList data structure is a resizable array (or performs resizing for us).

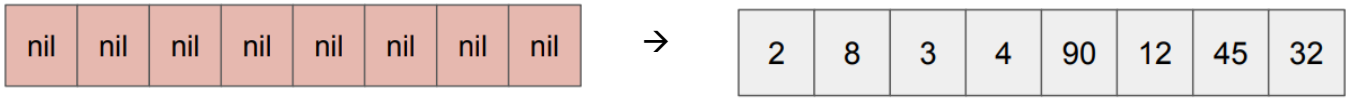


So let's disassemble these operations:

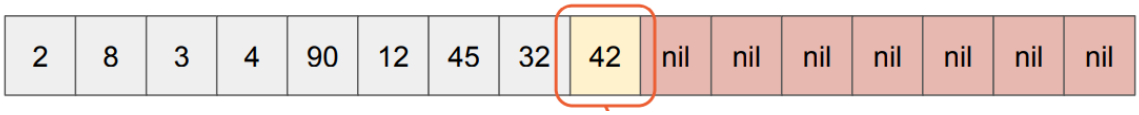
```
import java.util.ArrayList;
public class Example {
    public static int main(String[] b) {
        ArrayList<String> list = new ArrayList<String>();

        list.add("First String!");
        list.add("Second String!");
        list.add("Woof!!");

        list.remove(1);

        list.set(1, "Meow");
        System.out.println(list.get(0));
        System.out.println(list.get(1));

    }
}
```

Data Structures

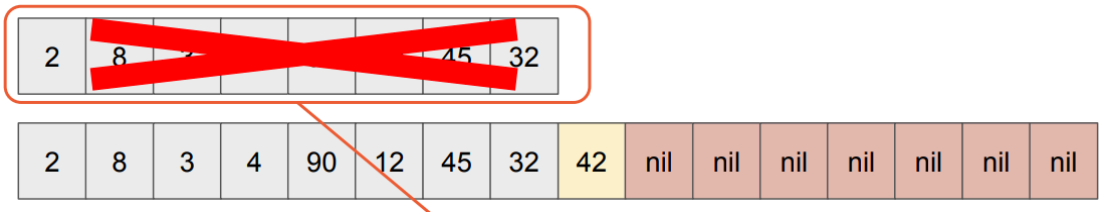
- So what is ArrayList doing that allows it to resize? → we will call it DynamicArray.
- We're given a starting value where all elements are null.
- When the .add() method is called, the DynamicArray will start adding elements into the Array.



- When all elements are used, a resize occurs since you cannot simply override an element → a new array is created that is double the size of the old one, containing no elements.
- The elements from the previous Array are then copied into the new one

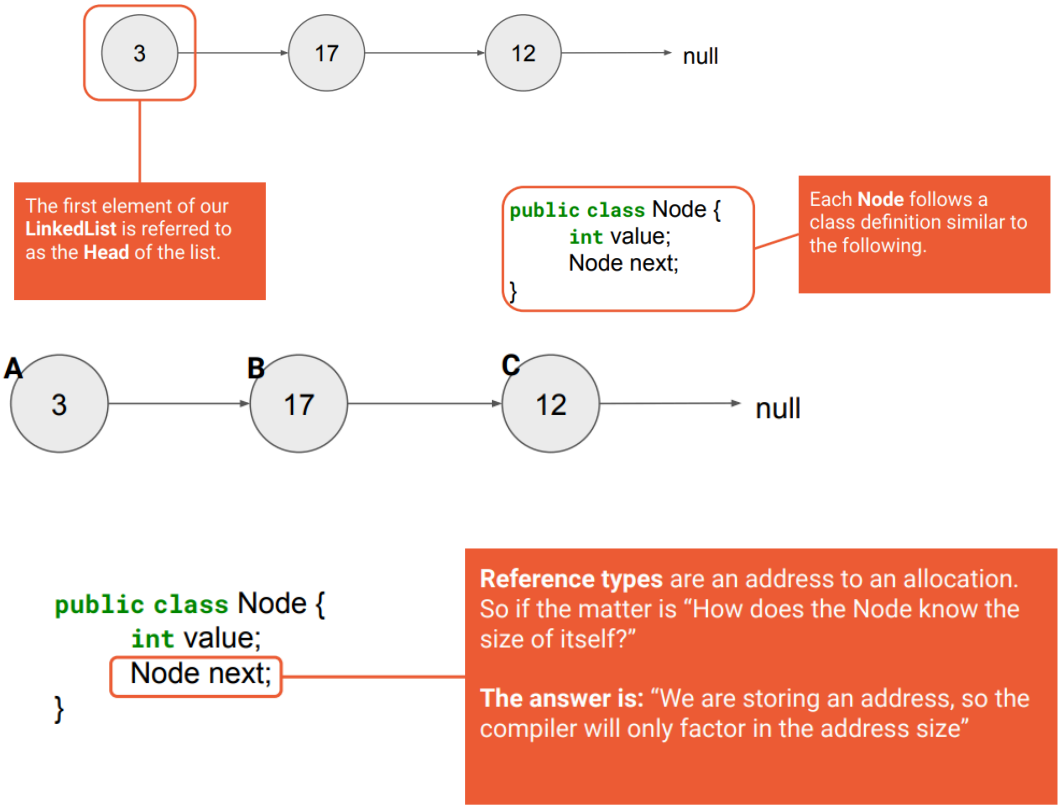


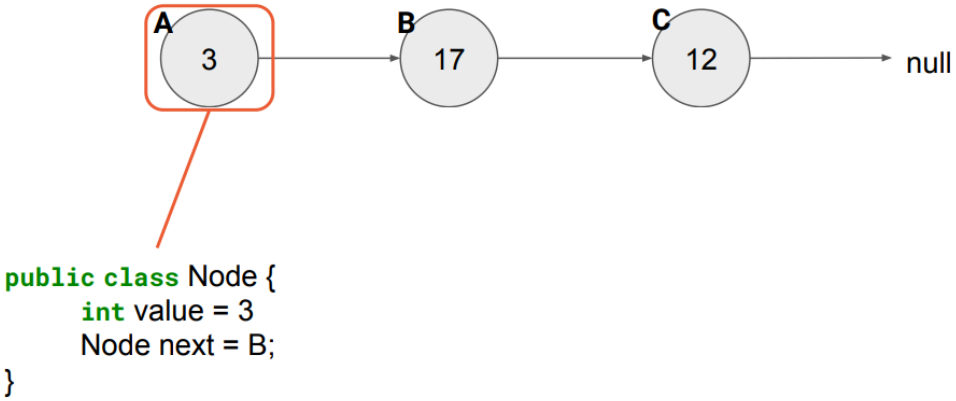
- After we have our new array, coped the elements across and added the new element, we lose the reference to the old array and it is collected by the garbage collector.



LinkedList

- Whilst the methods between a LinkedList and an ArrayList are practically the same, they differ fundamentally in their construction and behaviour.
- In contrast to an ArrayList, a LinkedList does not have an array containing all the elements stored. Instead, it chains elements and their values.
- Each element contains a value and a link to the next element. The link is commonly referred to as next and the elements within the LinkedList are commonly referred to as the Node.





- You will always encounter a problem which doesn't fit nicely and may require extra work.
- We can always combine collections within each other as they are just another type.
- E.g. ArrayList < ArrayLift < Integer >> dynamicMatrix;

Maps and Sets

- We want to be able to use non-integer objects for storage. This is where we bring in two different types that allow this → Map and Set.
- Not to be confused with a map method. A Map and Set provides a mapping of an object to a location where that element is stored.
- Types that are commonly used of this variety are:
    - → HashMap
    - → TreeMap
    - → HashSet
    - → TreeSet
- Map and Set have similar functionality in providing a mapping between an object and its location, but they do differ in a specific way.
- Maps are equivalent to the Python dictionary.

Syntax

```
Map<KeyType,ValueType> name = new
    Map<KeyType,ValueType>;

Set<Type> name = new Set<Type>;
```

So let's disassemble these operations:

```
import java.util.HashMap;
public class Example {
    public static int main(String[] b) {
        HashMap<String,Integer> seats = new HashMap<String,Integer>();

        seats.put("Bus", 30);
        seats.put("Car", 5);
        seats.put("Bike",1);
        seats.put("Truck", 2);

        seats.remove("Truck");

        System.out.println(seats.get("Bike"));
        System.out.println(seats.get("Car"));

    }
}
```

Instead of add() we have **put** instead. This is because **Key's** are **unique.**

So let's disassemble these operations:

```
import java.util.HashMap;
public class Example {
    public static int main(String[] b) {
        HashMap<String,Integer> seats = new HashMap<String,Integer>();

        seats.put("Bus", 30);
        seats.put("Car", 5);
        seats.put("Bike",1);
        seats.put("Truck", 2);

        seats.remove("Truck");

        System.out.println(seats.get("Bike"));
        System.out.println(seats.get("Car"));

    }
}
```

Remove elements based on the key.

Checked Operations

- We have a compiler to check that the types we are using are correct and we are not assigning data to the wrong variable. This is the referred to as a checked operation.
- This concept is important for collections since we have the concept of the reference and value types.
- With collection types, we only care about Reference types as they are the only type that can be used with the standard library collections (with generics).

Unchecked Operations

- If you are familiar with another language such as C, Python, JavaScript, Ruby (where either language is weakly typed or is dynamically typed, you may have encountered the situation where you provided arbitrary types to a list or array and you as a programmer know the order.
- However, this is an assumption and is considered an unsafe operation. In Java, the compiler likes to provide feedback that you are using types correctly in your code.
- It warns you when you have an Unchecked Operation.

So what does an unchecked warning look like?

We'll take a look at the syntax for List again.

Syntax:

```
List<Type> name = new ListType<Type>;
```

Why is this important for compiler?

Specifically, we omit the **Type** information from the list.

Using our previous example, what if we omit the type information and change what we add?

```
import java.util.ArrayList;
public class Example {
    public static int main(String[] b) {
        ArrayList list = new ArrayList();

        list.add("First String!");
        list.add(new Integer(5));
        list.add(new DefinitelyNotAString());

        list.remove(1);

        list.set(1, "Meow");

    }
}
```

By omitting the **type information** from the **ArrayList** we are able to add any object of any type to this collection.

What dangers are present when retrieving elements from this collection?

- We may use the wrong object (may assume it is a string when it is an integer)
- We have no idea what is stored there and therefore functionally useless.

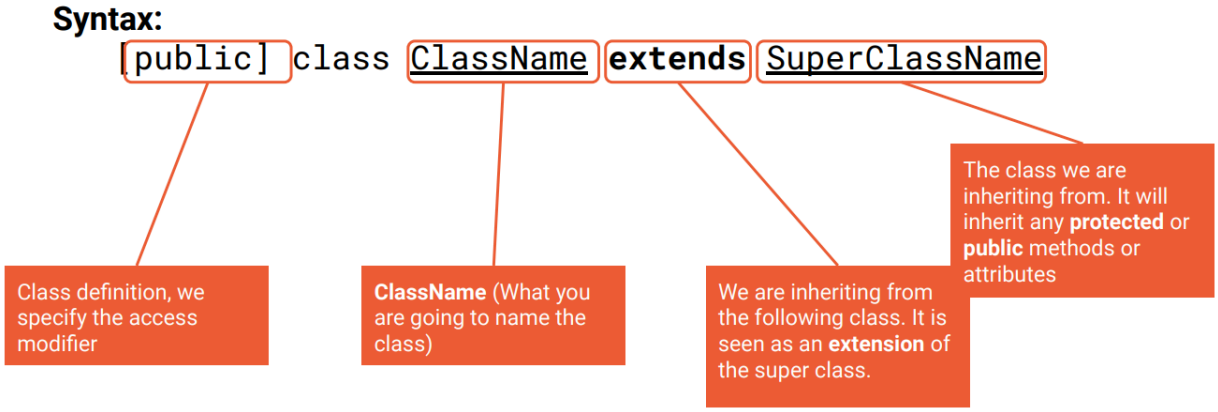
INFO1113 Week 5 Lecture – Class Inheritance and Overloading

Inheritance

- Inheritance is a significant concept of OOP → It allows reusability and changes to inherited methods between different types in hierarchy.
- What does inheritance offer?
    - → Attribute and method reusability
    - → Defining sub-type methods
    - → Overriding inherited methods
    - → Type information

How Does It Work?

- We will be introducing a new keyboard today called extends, allowing a class to inherit from another class.



- Part of our class declaration line allows for us to define what class we want to extend from. Once defined below, Dog type can also be used as a Canine type as it is just an extension of such type.

```
public class Dog extends Canine
```

Encapsulation

- The protected access modifier, like private, will not be accessible to other classes WITH the exception of inherited classes.
    - → Is only accessible within the class.
    - → Attributes and methods will be accessible by all subtypes
    - → Allows single definition of an attribute instead of multiple

```java
public class Bottle {

    protected String name;
    protected double width;
    protected double height;
    protected double depth;
    protected double litresFilled;


    public double volume() {
        return height*width*depth;
    }

}
```

```java
public class GlassBottle extends Bottle {

    protected String name;
    protected double width;
    protected double height;
    protected double depth;
    protected double litresFilled
    private boolean shattered = false;

    public void shatter() {
        System.out.println("We lost
        " + litresFilled + "Litres");
        litresFilled = 0;
        shattered = true;
    }

    public boolean isBroken() {
        return shattered;
    }
}
```

Able to refer to the attributes within the **subtypes** own methods.

- Subtypes will have access to any protected and public methods.
- All properties from the super class are inherited by the subclass, as if they were defined by the class itself.
- Assuming the default constructor is given to the superclass, the subclass does not need to define one.

Assuming the default constructor is given to the **superclass**, the **subclass** does not need to define one.

```java
public class Bottle {

    protected String name;
    protected double width;
    protected double height;
    protected double depth;
    protected double litresFilled;

    public Bottle() {
```

```java
public class GlassBottle extends Bottle {

    private boolean shattered = false;

    public void shatter() {
        shattered = true;
```

By default, when a **GlassBottle** object is created, it will refer to the **super** class's constructor.

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

    GlassBottle b = new GlassBottle();
    System.out.println(b.isBroken());
    System.out.println(b.name());
}
```

- In the example above, nothing was initialised, so all we get are default values.

```
public class Bottle {
    protected String name;
    protected double width;
    protected double height;
    protected double depth;
    protected double litresFilled;

    public Bottle() {
        this.name = "Basic Bottle";
        this.width = 10d;
        this.height = 10d;
        this.depth = 10d;
        this.litresFilled = 0;
    }

    public double volume() {
        return height*width*depth;
    }
}
```

Providing some values we can inspect the previous code segment

```
public class GlassBottle extends Bottle {

    private boolean shattered = false;

    public void shatter() {
        shattered = true;
    }

    public boolean isBroken() {
        return shattered;
    }
}
```

By default, when a **GlassBottle** object is created, it will refer to the **super** class's constructor.

```
public static void main(String[] args) {
    GlassBottle b = new GlassBottle();
    System.out.println(b.isBroken());
    System.out.println(b.name());
}
```

```
> java MyProgram
false
Basic Bottle
<program end>
```

We can see that even though we seemingly used the **GlassBottle** constructor.

- What if we were to define a constructor in the subtype?

```
public static void main(String[] args) {
    GlassBottle b = new GlassBottle();
    System.out.println(b.volume());
    System.out.println(b.name());
}
```

```
> java MyProgram
1000.0
Glass Bottle
<program end>
```

We can see that we called the **GlassBottle** constructor and it set the **name** to **Glass Bottle**.

- If we called GlassBottle(), how is volume returning 1000.0?
- What if we were to add a constructor with parameters?

Assuming the default constructor is given to the **superclass**, the **subclass** does not need to define one.

```
public class Bottle {
    protected String name;
    protected double width;
    protected double height;
    protected double depth;
    protected double litresFilled;

    public Bottle(String name, double width,
        double height, double depth) {
        this.name = name;
        this.width = width;
```

```
public class GlassBottle extends Bottle {

    public GlassBottle() {
        this.name = "Glass Bottle";
    }

    private boolean shattered = false;

    public void shatter() {
```

The **subclass must** invoke the **super** constructor. Using the **super** keyword, we are able to refer to inherited constructors and methods. However...

```
public static void main(String[] args) {
    GlassBottle b = new GlassBottle();
    System.out.println(b.volume());
    System.out.println(b.name());
}
```

How would the GlassBottle constructor be able to invoke the super constructor?

```
> javac MyProgram.java
./GlassBottle.java:5: error: constructor Bottle in class
Bottle cannot be applied to given types;
      public GlassBottle() {
                          ^
  required: String,double,double,double
  found: no arguments
  reason: actual and formal argument lists differ in length
1 error
```

- We use the super keyword to invoke the parent constructor.

Assuming the default constructor is given to the **superclass**, the **subclass** does not need to define one.

```
public class Bottle {
    protected String name;
    protected double width;
    protected double height;
    protected double depth;
    protected double litresFilled;

    public Bottle(String name, double width,
        double height, double depth) {
        this.name = name;
        this.width = width;
```

```
public class GlassBottle extends Bottle {

    public GlassBottle(String name, double
        width,double height,double depth){
        super(name, width, height, depth);

    }

    private boolean shattered = false;
```

Refers to **Bottle** constructor

The **subclass must** invoke the **super** constructor. Using the **super** keyword, we are able to refer to inherited constructors and methods. However...

```
public static void main(String[] args) {
    GlassBo
    System.
    System.out.println(b.name());
}
```

We could match the constructor of the parent type.

```
> javac MyProgram.java
./GlassBottle.java:5: error: constructor Bottle in class
Bottle cannot be applied to given types;
      public GlassBottle() {
                          ^
  required: String,double,double,double
  found: no arguments
  reason: actual and formal argument lists differ in length
1 error
```

Is-a and Has-a Relationship

- There are two types of relationships we will look at when it comes to inheritance:

  → Is-a relationship (extension)

  → Has-a relationship (composition)

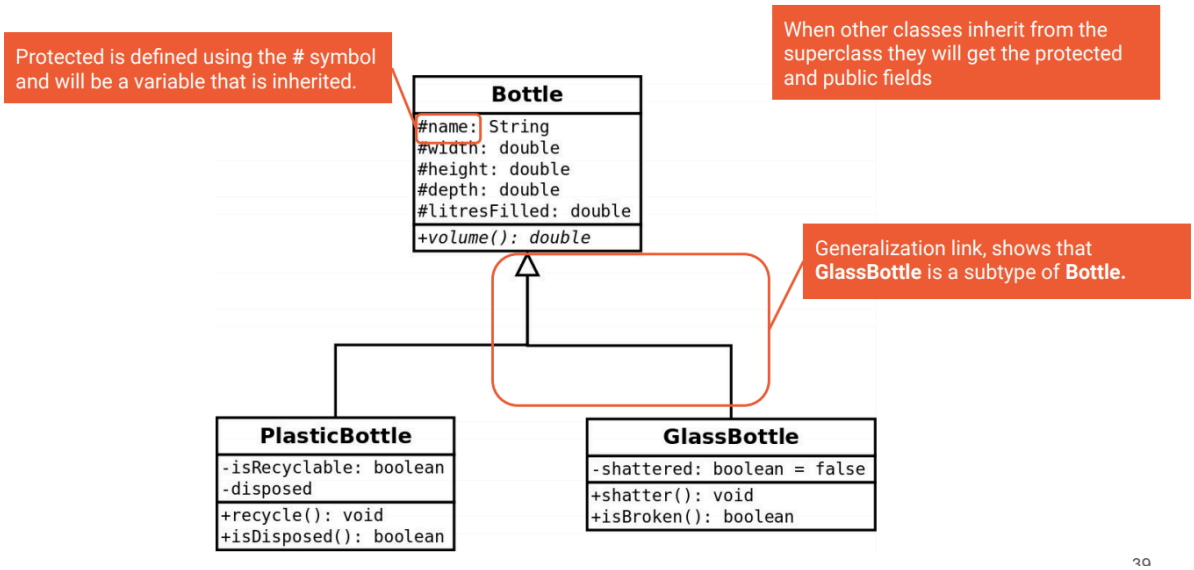
- With regard to class inheritance, we are considering the Is-a relationship for how a class is an extension of another but is also the other class.
- We have to be very certain with inheritance that any class that inherits from another IS A type of that class → there should be clear reasoning to extending the super class and that the types satisfy the relationship.
- Some instances where it makes sense:

  → Super class Cat with subclasses Panther, Lion, Tiger.

→ Super class Controller with subclasses Gamepad, Joystick and Powerglove

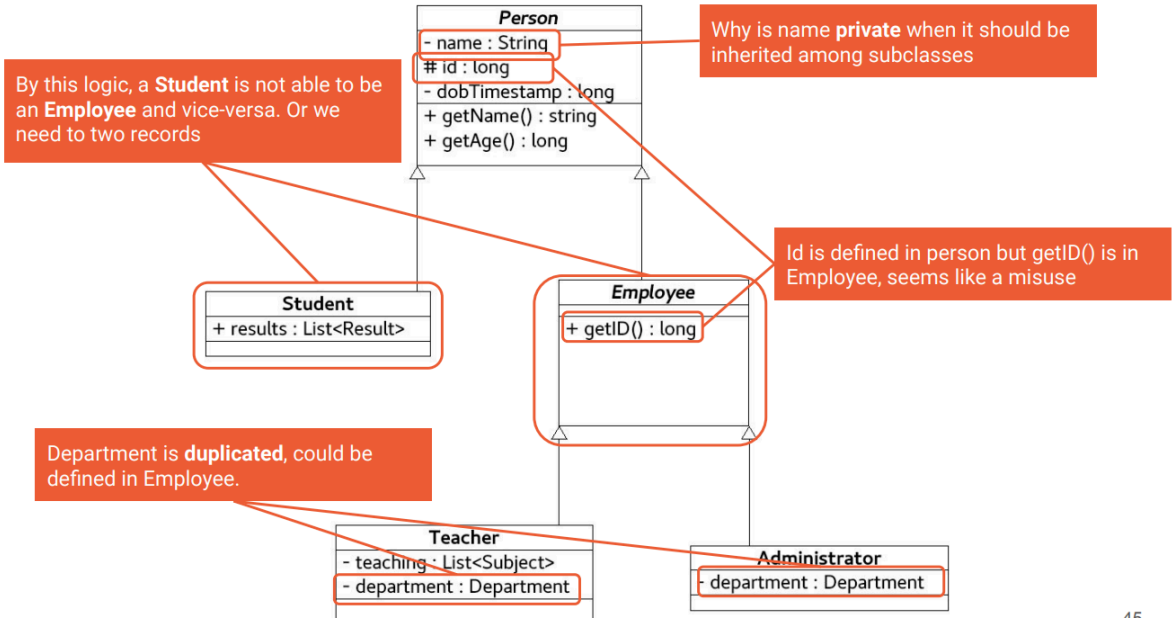
→ Super class Media with subclasses DVD, Book, Image

Let's examine the following UML Diagram.



Where Inheritance Fails/is Misused

Yes! Take a look at the following UML diagram.



Supertype and Subtype

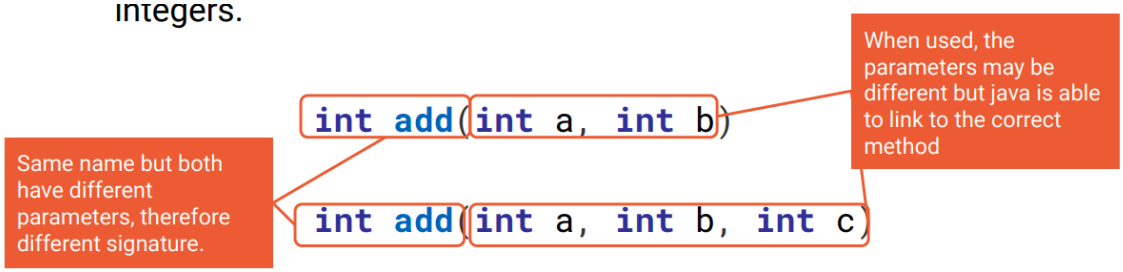
Some other factors to consider:

- Superclass does not know about its subclasses
- Private is not inherited, only protected and public
- Ensure when you use inheritance that you are very certain it will satisfy an is-a relationship.
- You can only inherit from one class.
- Within UML, inheritance is shown as a generalisation.
- You cannot use subclass properties through a superclass binding.
- Subtypes cannot be constructed using a supertype constructor e.g. SubType a = new SuperType();

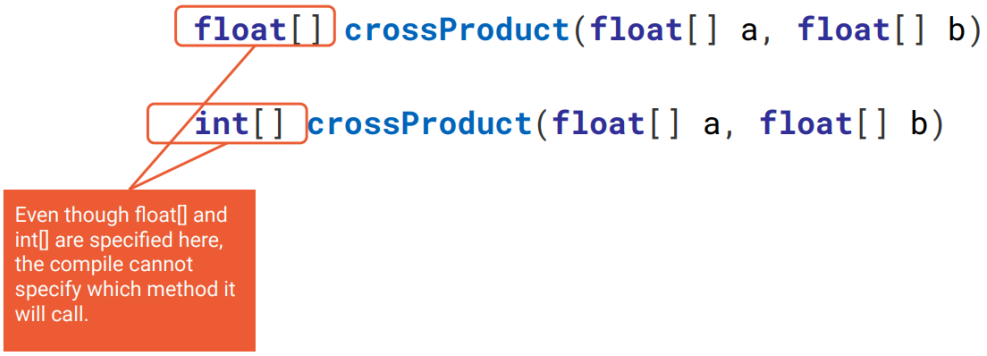
Overloading

- In regard to Java, we are able to use the same method name but with different method signature.
- Simply: we are able to define a method such as add and have a version that accepts two integers and another version that accepts three integers.

integers.



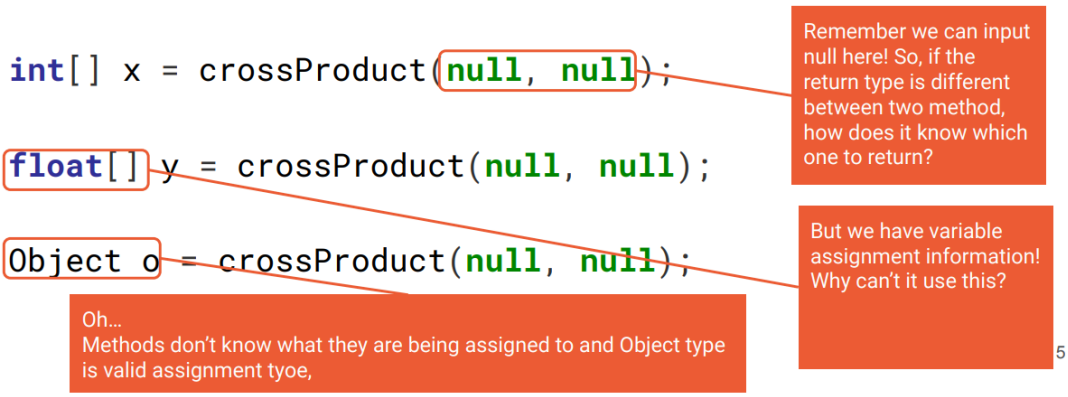
- We are unable to apply overloading if we have a different return type between the methods. The return type is not part of the method signature.



- Although we have type information here, all classes inherit from Object.
- Considering the following method calls using the two methods, assuming that they are correct.

```
float[] crossProduct(float[] a, float[] b)
int[] crossProduct(float[] a, float[] b)
```

Method calls:



Ambiguous Scenario

So let's consider the following method calls using the two methods and assume that they are correct.

```
int[] crossProduct(int[] a, int[] b)
int[] crossProduct(float[] a, float[] b)
```

By casting the reference to a certain type, the compiler can deduce what method to call

Method calls:

```
int[] x = crossProduct((int[])null, (int[])null);

int[] y = crossProduct((float[])null, (float[])null);
```

By casting float[] on the null references we can see it infer the method with floats as arguments

If we want to deal with ambiguous statements like this, we will need to cast, considering the **Object** example from before, we could actually be passing a real reference to an array

19

OTHERWISE:

Method calls:

```
int[] x = crossProduct(null, null);

int[] y = crossProduct(null, null);
```

Which method could it be calling?

The compiler would be unable to determine exactly what method is trying to be called and will throw an error.

```
> javac OverloadTest.java
OverloadTest.java:15: error: reference to crossProduct is
ambiguous
      int[] o = crossProduct(null, null);
                ^
  both method crossProduct(float[],float[]) in OverloadTest
and method crossProduct(int[],int[]) in OverloadTest match
1 error
```

18

Constructor Overloading

- We can observe the same overloading concept applied to constructors. This can be applied to both overloaded constructors within the same class as well as super constructors.
- We are able to also utilise certain constructors for other constructors if we have already defined that behaviour.

Let's take a look at the following class

```java
public class Person {

    private static int DEFAULT_AGE = 21;

    private String name;
    private int age;

    public Person() {
        name = "Jeff";
        age = DEFAULT_AGE;
    }

    public Person(String name) {
        this.name = name;
        this.age = DEFAULT_AGE;
    }

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

We can see that there are 3 different constructors. Within our own code we choose to call anyone, in fact we have already been doing this!

```java
public static void main(String[] args) {
    Person p1 = new Person(); //Jeff the default human!
    Person p2 = new Person("Janice");
    Person p3 = new Person("Dave", 32);
}
```

Since each constructor has a unique signature, we are able to utilise specific constructors by satisfying the correct types.

- Python isn't able to handle overloading constructors since functions in python are bound to variables, and we do not have type information or clean signatures in python.
- This could be overcome with default values for parameters and variable argument lists.

this Keyword

- The this keyword can play an important role in regards to constructors. It allows us to refer to the constructor within the context of a constructor.
- In particular, we can reduce the amount of code we write by reusing a constructor.

How could we use the this keyword in this example?

```java
public class Person {

    private static int DEFAULT_AGE = 21;

    private String name;
    private int age;

    public Person() {
        this("Jeff", DEFAULT_AGE);
    }

    public Person(String name) {
        this(name, DEFAULT_AGE);
    }

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

By using the **this** keyword, we are able to eliminate 2 lines from the other constructors by using the last one.

Calls this constructor

super Keyword

- We will be exploring inheritance with constructor overloading and method overriding and how we are able to utilise inherited behaviour in our program.
- We will show how we are able to access elements through the super keyword.

How could we use the this keyword in this example?

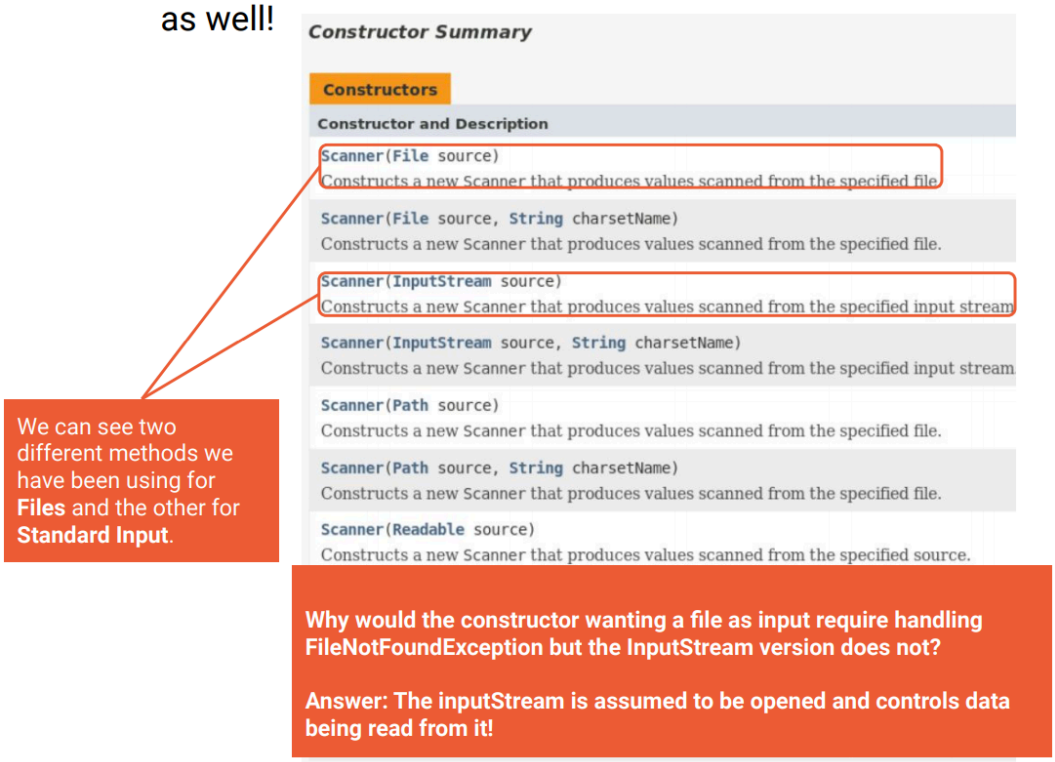
```java
public class Person {

    private static int DEFAULT_AGE = 21;

    private String name;
    private int age;

    public Person() {
        this("Jeff", DEFAULT_AGE);
    }

    public Person(String name) {
        this(name, DEFAULT_AGE);
    }

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
    //<snipped getName(), setName(), getAge()
}
```

```java
public class Employee extends Person {

    private long employeeId;
    private long departmentId;

    public Employee(String name, int age,
        long departmentId, long employeeId) {

        super(name, age);
        this.departmentId = departmentId;
        this.employeeId = employeeId;
    }
    //<snipped other methods
}
```

We are able to specify the constructor we want to invoke and set attributes for the object.

Java I/O

Overloading is not just restricted to methods, we are able to apply it to constructors. This is evident within the standard library itself as well!



- We are going to examine how inheritance works within the Java API via the java.io package.
- Java has main superclasses broken into:
    - → Reader
    - → Writer
    - → InputStream
    - → OutputStream
- With these super classes, they dictate the behaviour of how data is read and the scenario we will employ them in.
- This can be further generalised that:
    - → Reader and Writer classes are Character Stream classes.
    - → InputStream and OutputStream classes are Byte Stream classes.

IO Classes

When inspecting the java.io classes. We can start seeing a pattern in how they have given responsibility to each class.

FileWriter -> Character stream on files

FileOutputStream -> Byte stream on files

Considering IO is not strictly used for file they will utilise similar behaviour from inherited classes for their needs.

BufferedInputStream	A `BufferedInputStream` adds functionality to another input stream-namely, the ability to buffer the input and to support the `mark` and `reset` methods.
BufferedOutputStream	The class implements a buffered output stream.
BufferedReader	Reads text from a character-input stream, buffering characters so as to provide for the efficient reading of characters, arrays, and lines.
BufferedWriter	Writes text to a character-output stream, buffering characters so as to provide for the efficient writing of single characters, arrays, and strings.
ByteArrayInputStream	A `ByteArrayInputStream` contains an internal buffer that contains bytes that may be read from the stream.
ByteArrayOutputStream	This class implements an output stream in which the data is written into a byte array.
CharArrayReader	This class implements a character buffer that can be used as a character-input stream.
CharArrayWriter	This class implements a character buffer that can be used as an Writer.
Console	Methods to access the character-based console device, if any, associated with the current Java virtual machine.
DataInputStream	A data input stream lets an application read primitive Java data types from an underlying input stream in a machine-independent way.
DataOutputStream	A data output stream lets an application write primitive Java data types to an output stream in a portable way.
File	An abstract representation of file and directory pathnames.
FileDescriptor	Instances of the file descriptor class serve as an opaque handle to the underlying machine-specific structure representing an open file, an open socket, or another source or sink of bytes.
FileInputStream	A `FileInputStream` obtains input bytes from a file in a file system.
FileOutputStream	A file output stream is an output stream for writing data to a `File` or to a `FileDescriptor`.
FilePermission	This class represents access to a file or directory.
FileReader	Convenience class for reading character files.
FileWriter	Convenience class for writing character files.
FilterInputStream	A `FilterInputStream` contains some other input stream, which it uses as its basic source of data, possibly transforming the data along the way or providing additional functionality.
FilterOutputStream	This class is the superclass of all classes that filter output streams.
FilterReader	Abstract class for reading filtered character streams.
FilterWriter	Abstract class for writing filtered character streams.
InputStream	This abstract class is the superclass of all classes representing an input stream of bytes.
InputStreamReader	An InputStreamReader is a bridge from byte streams to character streams: It reads bytes and decodes them into characters using a specified `charset`.
LineNumberInputStream	Deprecated *This class incorrectly assumes that bytes adequately represent characters.*
LineNumberReader	A buffered character-input stream that keeps track of line numbers.
ObjectInputStream	An ObjectInputStream deserializes primitive data and objects previously written using an ObjectOutputStream.
ObjectInputStream.GetField	Provide access to the persistent fields read from the input stream.
ObjectOutputStream	An ObjectOutputStream writes primitive data types and graphs of Java objects to an OutputStream.
ObjectOutputStream.PutField	Provide programmatic access to the persistent fields to be written to ObjectOutput.
ObjectStreamClass	Serialization's descriptor for classes.
ObjectStreamField	A description of a Serializable field from a Serializable class.
OutputStream	This abstract class is the superclass of all classes representing an output stream of bytes.
OutputStreamWriter	An OutputStreamWriter is a bridge from character streams to byte streams: Characters written to it are encoded into bytes using a specified `charset`.

java.io package summary, Oracle (https://docs.oracle.com/javase/7/docs/api/java/io/package-summary.html)

Exceptions

- Exceptions are throwable methods → some constructors and methods require wrapping around a try-catch block.
- However, not all exceptions require this → like with IO, Exceptions have an inheritance hierarchy and contain specific error messages to inform the programmer of the error that has occurred.

Class Exception

java.lang.Object
  java.lang.Throwable
    java.lang.Exception

All Implemented Interfaces:

Serializable

Direct Known Subclasses:

AclNotFoundException, ActivationException, AlreadyBoundException, ApplicationException, AWTException, BackingStoreException, BadAttributeValueExpException, BadBinaryOpValueExpException, BadLocationException, BadStringOperationException, BrokenBarrierException, CertificateException, CloneNotSupportedException, DataFormatException, DatatypeConfigurationException, DestroyFailedException, ExecutionException, ExpandVetoException, FontFormatException, GeneralSecurityException, GSSException, IllegalClassFormatException, InterruptedException, IntrospectionException, InvalidApplicationException, InvalidMidiDataException, InvalidPreferencesFormatException, InvalidTargetObjectTypeException, IOException, JAXBException, JMException, KeySelectorException, LambdaConversionException, LastOwnerException, LineUnavailableException, MarshalException, MidiUnavailableException, MimeTypeParseException, MimeTypeParseException, NamingException, NoninvertibleTransformException, NotBoundException, NotOwnerException, ParseException, ParserConfigurationException, PrinterException, PrintException, PrivilegedActionException, PropertyVetoException, ReflectiveOperationException, RefreshFailedException, RemarshalException, RuntimeException, SAXException, ScriptException, ServerNotActiveException, SOAPException, SQLException, TimeoutException, TooManyListenersException, TransformerException, TransformException, UnmodifiableClassException, UnsupportedAudioFileException, UnsupportedCallbackException, UnsupportedFlavorException, UnsupportedLookAndFeelException, URIReferenceException, URISyntaxException, UserException, XAException, XMLParseException, XMLSignatureException, XMLStreamException, XPathException

- Two major Exception classes exist that dictate how the rest operate.

  → Exception and any subclasses (with the exception of RuntimeException) is a checked exception → when a method can throw the exception, the programmer must handle it using a try-catch block.

  → RuntimeException and any subclasses, is an unchecked exception. The programmer does not need to handle this case but can catch if they want to – this should be a case where the program should crash.

Let's examine the following

```
public void imGonnaCrash() throws Exception {
    throw new Exception("Definitely crashing!");
}
```

Where we throw the exception, typically this is in some kind of if statement.

Since the method can throw a **checked exception** we are required it to handle it when we call it.

Within our main method we **cannot proceed** with the following.

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

We are **forced** to catch it by the compiler;.

```
public static void main(String[] args) {
    try {
        imGonnaCrash();
    } catch(Exception e) {
        e.printStackTrace();
    }
}
```

Where the compiler will **not** force the programmer to handle a **RuntimeException.**

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

INFO1113 Week 6 Lecture – Abstract Classes, Interfaces, Polymorphism and Packaging

Abstract Classes

- Although similar to a concrete class, an abstract class cannot be instantiated. It can define methods and attributes which cannot be inherited from nor inherit from super types.
- However, they can also enforce a method implementation for subtypes.

Why Would We Use One?

- The main case for abstract types is that we have some type that we do not want instantiated but is a generalisation of many other types.
- E.g. Shape is a generalisation of Triangle, Square, Circle etc. but we don't have a concrete instance of Shape.
- E.g. Furniture is a generalisation of Chair, Sofa, Table and Desk.

What Can We Still Do?

We still are able to specify:

- Constructors
- Define methods (static and instance)
- Attributes
- Use all the access modifiers
- Everything a regular class can do except we cannot instantiate the class but we can specify the methods subtypes must define.

Declaration of An Abstract Class

- We define an abstract class by using the abstract keyword. This immediately marks the class as abstract and we don't need anything more.

Syntax:

```
[modifier] abstract class ClassName
```

Example:

```
public abstract class Furniture
```

- Since it is marked as abstract, the compiler will refuse to allow this type of instantiation.
- There is a little more at work here though when we mark something as abstract, because we can mark methods as being abstract as well.

```
public abstract void stack(Furniture f);
```

```
> javac FurnitureStore.java
FurnitureStore.java:22: error: Furniture
is abstract; cannot be instantiated
    Furniture f = new Furniture("Table");
                      ^
1 error
<program end>
```

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

public abstract class Furniture {

    private String name;
    private List<Part> parts;

    public Furniture(String name) {
        this.name = name;
        this.parts = new ArrayList<Part>();
    }

    public void addPart(Part p) {
        parts.add(p);
    }

    public abstract void stack(Furniture f);
}
```

Notice we have declared an **abstract** method.

```java
public class WoodChair extends Furniture {

    public WoodChair() {
        super("WoodChair");
    }

}
```

However, in this class we have not defined the method **stack**.

```
> javac FurnitureStore.java
WoodChair.java:1: error: WoodChair is not abstract
and does not override abstract method
stack(Furniture) in Furniture
public class WoodChair extends Furniture {
       ^
1 error
```

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

public abstract class Furniture {

    private String name;
    private List<Part> parts;

    public Furniture(String name) {
        this.name = name;
        this.parts = new ArrayList<Part>();
    }

    public void addPart(Part p) {
        parts.add(p);
    }

    public abstract void stack(Furniture f);
}
```

```java
public class WoodChair extends Furniture {

    public WoodChair() {
        super("WoodChair");
    }

    public void stack(Furniture f) {
        System.out.println("Don't put
          furniture on chairs!");
    }
```

Now we have defined the method **stack** in the subclass.

```
> javac FurnitureStore.java
>
```

```java
public class FurnitureStore {

    public static void main(String[] args) {
        WoodChair f = new WoodChair();
        f.stack(new WoodChair());
    }
}
```

We can now declare and invoke stack through **WoodChair** class.

```
> java FurnitureStore
Don't put furniture on chairs!
```

- We can even bind it to Furniture type and invoke stack which will call the subtype's method.

Abstract Methods

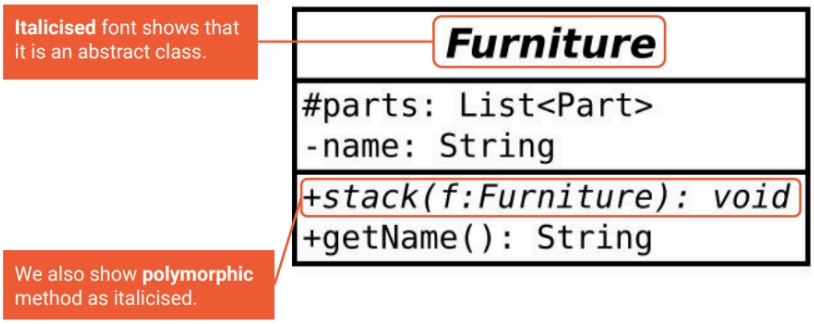
- We are able to declare an abstract method in only abstract classes → when we declare an abstract method, we do not define a method body (the logic of the method).

```java
public abstract void stack(Furniture f);
```

- If the class should not be instantiated or behaviour is defined by the subtypes and not the super type, the class should be non-instantiable.

Abstract Classes and UML

- Within a UML class diagram, we can illustrate abstract classes with the following.



- This is convention for UML 2.

Interfaces

- We will be introducing a new keyword implements.
- Interfaces share a similarity with Abstract Classes in that they declare methods that a class must implement, and they cannot be instantiated → an interface is NOT a class.
- However, unlike classes, they can be implement by classes as many times as they like.
- We are not bound to implementing a single interface, we can implement multiple interfaces.

Simply we are able to define an **interface** by using the **interface keyword**. Afterwards we will need to define a few

Syntax:

```
[modifier] interface InterfaceName
```

Example:

```
public interface Swim {
    public void swim();
    public void dive();
}

public class Dog implements Swim
```

To be clear, an **interface** is **not a class.** It defines a group a methods for implementers to define.

Since a **Dog** class **implements** the **Swim** interface it will need to define the methods for **Swim.**

Why Would We Want Interfaces?

- Interfaces:
  - → Cannot specify any attributes, only methods.
  - → Do not (typically) provide a method definition.
  - → Cannot instantiate them.
  - → Can be implemented multiple times.
- From an application design perspective, we need to consider how we can use interfaces and where they are appropriate.

Example

So let's take a look at the following example (WHOA!)

```
public interface Move {
    public void move(double hours);
}
```

We have defined our **Interface Move** that will be implemented by **Dog** and **Dolphin**.

```
public class Dog implements Move {
    private String region; //Water or Land
    private final double LAND_MOVEMENT_SPEED_KMH = 50.0;
    private final double WATER_MOVEMENT_SPEED_KMH = 10.0;
    private double kmTravelled = 0.0;

    public Dog(String region) {
        this.region = region;
    }

    public void move(double hours) {
        if(region.equals("water")) {
            kmTravelled += (WATER_MOVEMENT_SPEED_KMH *hours);
        } else if(region.equals("land")) {
            kmTravelled += (LAND_MOVEMENT_SPEED_KMH * hours);
        }
    }

    public double getKMTravelled() {
        return kmTravelled;
    }
}
```

```
public class Dolphin implements Move {
    private String region; //Water or Land
    private final double LAND_MOVEMENT_SPEED_KMH = 1.0;
    private final double WATER_MOVEMENT_SPEED_KMH = 60.0;
    private double kmTravelled = 0.0;

    public Dolphin(String region) {
        this.region = region;
    }

    public void move(double hours) {
        if(region.equals("water")) {
            kmTravelled += (WATER_MOVEMENT_SPEED_KMH * hours);
        } else if(region.equals("land")) {
            kmTravelled += (LAND_MOVEMENT_SPEED_KMH * hours);
        }
    }

    KMTravelled() {
    elled;
```

Since they both implement **Move** interface, we can treat them as a **Move type.**

They both have a similar implementation but **their** land and water movement speed is different. We could change it completely between the two implementations.

```java
public class MovingAnimals {
    public static void main(String[] args) {
        Dog dog = new Dog("land");
        Dolphin dolphin = new Dolphin("land");
        Move[] movingAnimals = {dog, dolphin};

        for(Move m : movingAnimals) {
            m.move(1.0);
        }

        System.out.println(dog.getKMTravelled());
        System.out.println(dolphin.getKMTravelled());
    }
}
```

We can create an **Move[]** array and add both **dog** and **dolphin** types to it.
Because they are of type Move.

```java
public class MovingAnimals {
    public static void main(String[] args) {
        Dog dog = new Dog("land");
        Dolphin dolphin = new Dolphin("land");
        Move[] movingAnimals = {dog, dolphin};

        for(Move m : movingAnimals) {
            m.move(1.0);
        }

        System.out.println(dog.getKMTravelled());
        System.out.println(dolphin.getKMTravelled());
    }
}
```

If they of **type Move** we are guaranteed to be able to use **move()** method.

Reflecting on your experience with **Python** or **Javascript. Why do we not need to treat them as a certain type in those languages? Simply: They are not statically typed.**

```java
public class MovingAnimals {
    public static void main(String[] args) {
        Dog dog = new Dog("land");
        Dolphin dolphin = new Dolphin("land");
        Move[] movingAnimals = {dog, dolphin};

        for(Move m : movingAnimals) {
            m.move(1.0);
        }

        System.out.println(dog.getKMTravelled());
        System.out.println(dolphin.getKMTravelled());
    }
}
```

```
> java MovingAnimals
50.0
1.0
<program end>
```

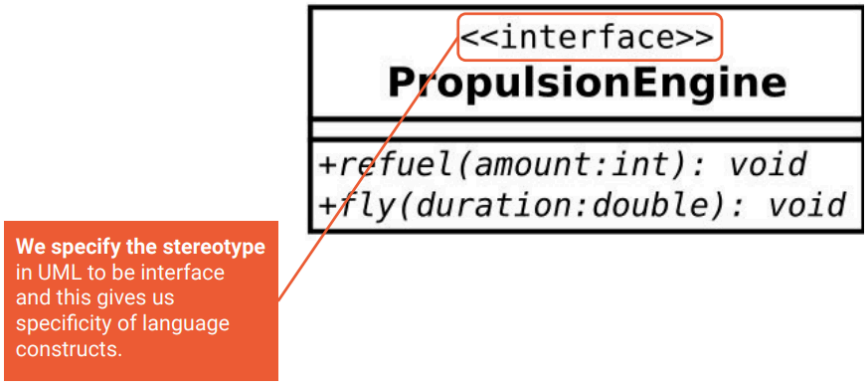
We can then see that **move()** has changed an **internal** travelled variable.

Note: Interfaces

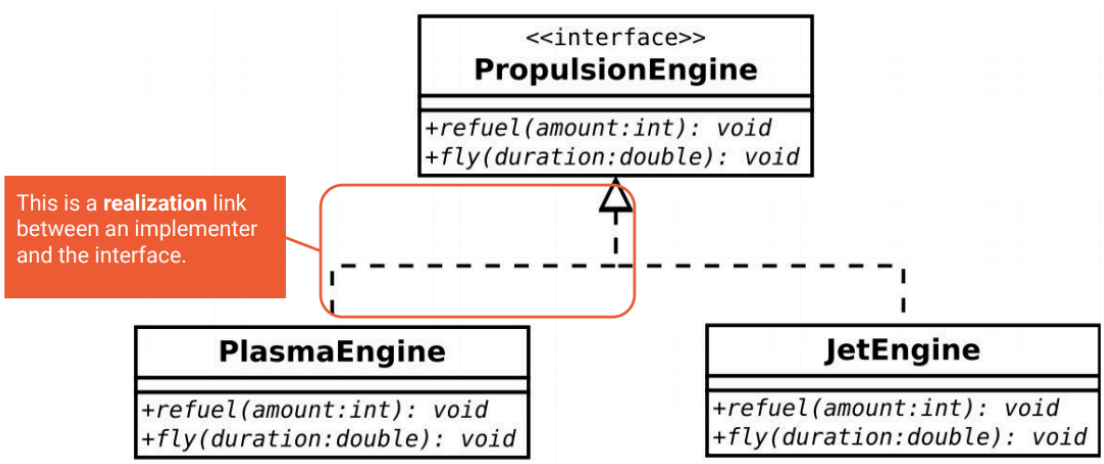
- We can have variables in an interface. However, the variables are:
  - → Static (they belong to the interface).
  - → Constant (have the final modifier applied to them).
- Therefore, we cannot use them for instances.

UML

- Just like abstract classes, we can represent an interface within UML → however it is slightly different than others.
- However, the relationship link is different than that of a class.



- Italicised font shows that it is a polymorphic method.



Default Methods

- Prior to Java 8, interfaces just specified the method declaration and never a default method.
- Default methods are a feature of java 8 and it can be difficult to see their use case. In particular, interfaces do not have access to any instance variables and therefore, it may be difficult to see how a default method can be used for reducing complexity of classes.
- We are able to depend on the implementation of interface methods that have declared in the interface. We are then able to utilise the definition given by the concrete type.

Simply we are able to define an **interface** by using the **interface keyword**. Afterwards we will need to define a few

Syntax:

```
[modifier] default <returntype> MethodName([parameters])
```

Example:

```
private default void swim();
```

Note: the **default** keyword presence is only noted in default methods and **switch** statements.

- The liquid container behaviour allows us to fill the container and pour the liquid out of it.
- Both CoffeeShot and CoffeeCup implement this behaviour through the interface.

```java
interface LiquidContainer {
    public void pour(double litres);
    public void pourInto(LiquidContainer container, double litres);
    public void fill(double litres);
}
```

```java
public class CoffeeShot implements LiquidContainer {
    private double litres = 0;
    public void pour(double litres) {
        this.litres -= litres;
    }
    public void pourInto(LiquidContainer container, double litres) {
        if(container != null) {
            pour(litres);
            container.fill(litres);
        } else {
            pour(litres);
        }
    }
    public void fill(double litres) {
        this.litres += litres;
    }
}
```

```java
public class CoffeeCup implements LiquidContainer {
    private double litres = 0;
    public void pour(double litres) {
        this.litres -= litres;
    }
    public void pourInto(LiquidContainer container,
        double litres) {
        if(container != null) {
            pour(litres);
            container.fill(litres);
        } else {
            pour(litres);
        }
    }
    public void fill(double litres) {
        this.litres += litres;
    }
}
```

We can observe that **pour** and **fill** interact with **instance** variables and this makes them a very poor candidate in being generalised to a **default method**.

- HOWEVER, pourInto contains certain properties that allow it to be ideal in this even, since it doesn't interact with instance methods directly, and uses interface methods.
- We can also see the logic is duplicated between the two classes as well. Considering we want to eliminate duplication, the way we can remove this is through a default method.

```java
interface LiquidContainer {
    public void pour(double litres);

    public default pourInto(LiquidContainer container, double litres) {
        if(container != null) {
            pour(litres);
            container.fill(litres);
        } else {
            pour(litres);
        }
    }

    public void fill(double litres);
}
```

We can move it to the interface and it the method is implemented and can be used by each class that implements **LiquidContainer.**

```java
public class CoffeeShot implements LiquidContainer {
    private double litres = 0;
    public void pour(double litres) {
        this.litres -= litres;
    }

    public void fill(double litres) {
        this.litres += litres;
    }
}
```

```java
public class CoffeeCup implements LiquidContainer {
    private double litres = 0;
    public void pour(double litres) {
        this.litres -= litres;
    }

    public void fill(double litres) {
        this.litres += litres;
    }
}
```

Overriding

- Overriding allows us to specify a subtype implementation of the method.
- Overriding applies for Classes, Abstract Classes, and Interfaces.
- We are able to have a specific implementation that will always refer to the subtype implementation, or narrow down to the last type that implemented it in the hierarchy.
- We are able to override inherited methods and replace them with a subtype specific definition.

Polymorphism

- Polymorphism allows us to assert the use of methods specified in an inherited type through other types. We have already seen polymorphism employed in a number of ways.
  - → Through interfaces and abstract classes.
  - → Method overloading
  - → Method overriding
- We are able to infer what methods we have access through the type information associated with the object.

```java
public class Cat {

    private String name;

    public Cat(String name) {
        this.name = name;
    }

    public void makeNoise() {
        System.out.println("Meow!");
    }

}
```

```java
public class DomesticCat {

    public DomesticCat(String name) {
        super(name);
    }

}
```

```java
public class Lion {

    public Lion(String name) {
        super(name);
    }

    public void makeNoise() {
        System.out.println("Roar!");
    }

}
```

- We have already defined the makeNoise() method that the majority of cats will make in the super class.
- We can see that the DomesticCat does not attempt to override this as it is appropriate for the class.
- However, Lions 'roar' instead, and should therefore make a different noise → despite this, we can treat all three classes under the superclass.

```java
public static void main(String[] args) {
    Cat[] cats = {new Cat("Felix"),
        new DomesticCat("Garfield"),
        new Lion("Simba")
    };

    for(Cat c : cats) {
        c.makeNoise();
    }
}
```

```
> java CatProgram
Meow!
Meow!
Roar!
```

Final Classes and Methods

- We have the power to inherit from classes but this could potentially open our code up for abuse by other inheriting from classes that shouldn't be done.
- We will introduce a way of stopping classes from being inherited, and methods from being overridden.

Similar to the **leaf** class we are able to use the final qualifier as part of the method declaration. Similar to **final** with classes, this infers that the method cannot be overridden.

Syntax:

```
[modifier] final <returntype> MethodName([parameters])
```

Example:

```
private final void swim();
```

Note: the **default** keyword presence is only noted in default methods and **switch** statements.

- When assembling a library, we want to ensure that users of the library cannot break it in ways that may induce undefined behaviour. This allows us to also differentiate between an error within a library and a user's code.

```
public class Person {
    private String name;

    public Person(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

}
```

```
public class FakePerson extends Person {

    public FakePerson(String name) {
        super(name);
    }

    public String getName() {
        System.out.println("Infinite Loop! Yeah");
        int i = 0;
        while(i < 10) {}
        return null;
    }
}
```

- We have defined two classes, Person and FakePerson, where the latter overrides the getName() method with its own implementation.
- However, we can see the implementation is malicious and can allow for execution of arbitrary code.
- We are able to PREVENT overriding of the method by the subtype by using final → prevented by the compiler.

```
public final String getName() {
    return name;
```

```
public class PersonProgram {
    public static void sayName(Person p) {
        String name = p.getName();
        System.out.println(name);
    }

    public static void main(String[] args) {
        Person p = new Person("Jimmy");
        FakePerson f = new FakePerson("Mr Evil");
        sayName(p);
        sayName(f);
    }
}
```

```
> java PersonProgram
PersonProgram.java:20: error: getName() in
FakePerson cannot override getName() in
Person
    public String getName() {
                  ^
  overridden method is final
1 error
```

Final Classes

- It is best to not be confused with a constant class and consider all attributes are read-only once initialised. The final qualifier specifies that the class is a leaf class (an endpoint to the hierarchy chain).

Syntax:

```
[modifier] final class ClassName
```

Example:

```
public final class Person
```

Scenarios for Leaf Classes

- We may not want to allow any inheritance due to how the classes have been designed or it may not make logical sense to extend from such a class e.g.
  - → inheritance hierarchy of military ranks, should we allow the highest rank to be extended?
  - → when modelling a Monarch, would we extend from the type with the largest or lowest authority.
  - → A User and Administrator, should the Administrator inherit from User or vice versa.

Classpath

- A classpath defines a set of directories exposed to our program. It will allow us to use libraries constructed by others.
- We are able to cleanly separate and structure our code into different directories and refer to different segments as if it was in the same directory.
- Given a directory that will allow us to store our class file, we will be able to use them directly within our project.

```
> javac -cp .:<Your directory or jar file here>[:<more>]
```

- You will need to ensure when running your application, that your program has access to any class it depending on during compilation → otherwise, your application will not have access to the class files required.

Issues?

- Potentially, two classes can exist with the same name. If this is the case, we have competing classes and we are unable to compile the program with this ambiguity → rare with small programs.

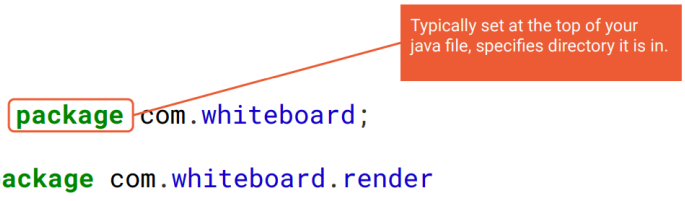
Packages

- Java defines a package keyword which will outline to the class which part of the package it resides in. It will self-verify on compilation if it exists within the package.

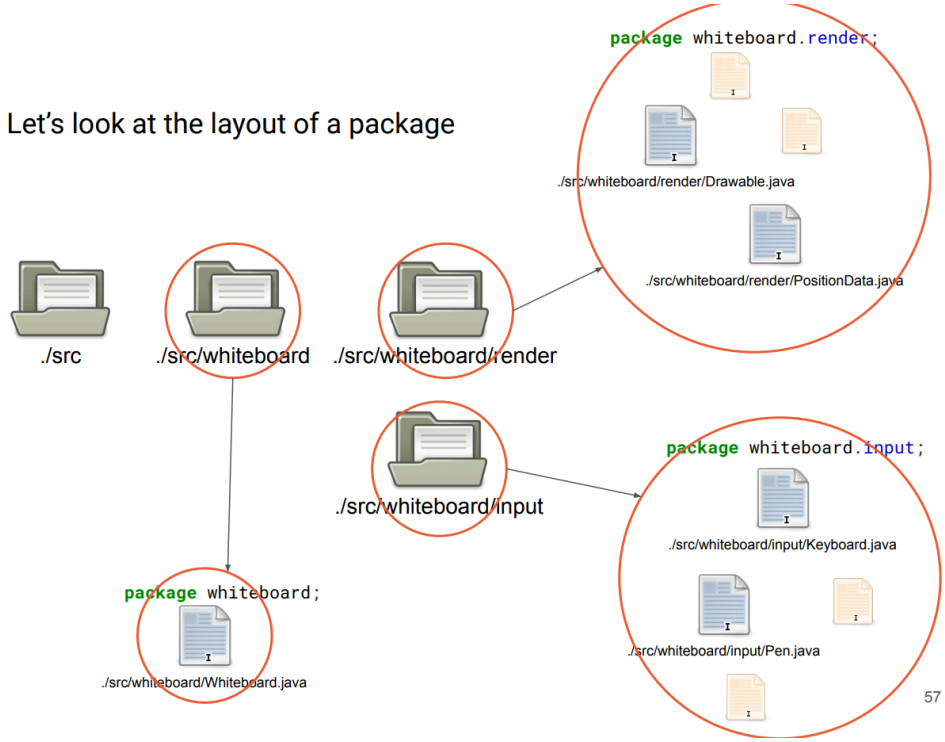
Syntax:

```
package <identifier>[.<nested ident>[...]]
```

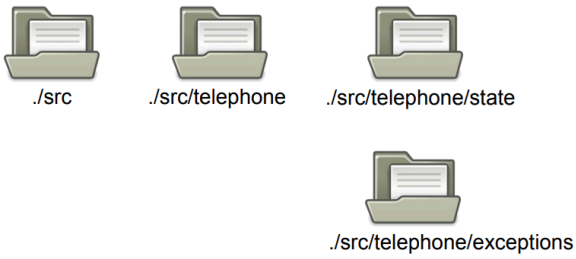
Example:



- However, there is a drawback with this. As noted before, we do not have the ability to distinguish between locations that may have classes of the same name → these packages are being considered under the default package name.

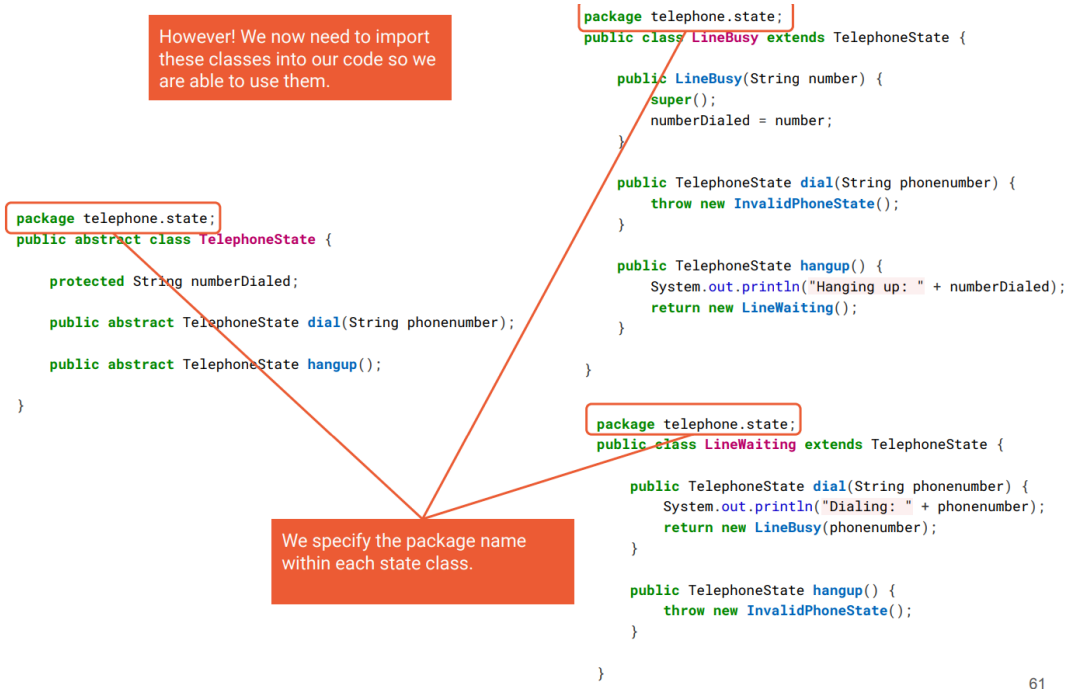


- Thus imagine we have loud out the package as following:




```java
package telephone;
public class Telephone {

    private TelephoneState state;

    public Telephone() {
        state = new LineWaiting();
    }

    public void dial(String phonenumber) {
        state = state.dial(phonenumber);
    }

    public void hangup() {
        state = state.hangup();
    }

    public static void main(String[] args) {
        Telephone phone = new Telephone();
        phone.dial("12341234");
        phone.hangup();
    }

}
```

We specify above our above classes and typically above majority of our code, the package name for the file. It is **best** practice to put the source file within a folder of the **same name.**



How Could We Create an Archive?

- Java provides an archiving format that allows you to compress the files you want to export and distribute to something else.
- This kind of format is similar to other OS/Package manager specific formats such as .dmg, .apk, .xdg and .deb.
- .jar Manifest files provide a simple description of requirements your archive files need.
- A common setting is providing an Application Entry point for your .jar file.
- By default, creating an archive file will only index the files you have added to it. It will not know what .class file you want to execute. You will need to specify that by hand.

To create an archive file, you will need to utilise the **jar** command. We are able to store any kind of data within a java archive but its typical case is bundling and packaging of libraries and applications.

```
> jar cf MyProgram.jar <list of files>
```

Specifies the create and file flag for the **jar** program.

We specify the Jar file to produce and input .class files to be included in the archive.

INFO1113 Week 7 Lecture – Generics, Collection Interfaces and Type Checking

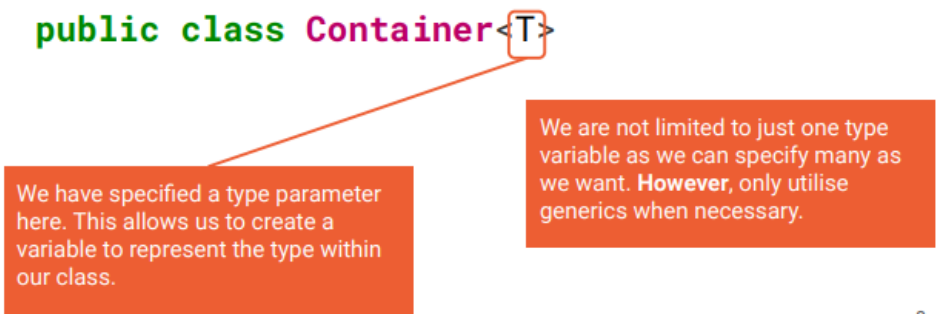
Generics

- Generics give us the ability to handle multiple different types without needing to rewrite the same code.
- The advantages of generics include stronger type checks at compile time, elimination of casts, and that they enable programmers to implement generic algorithms.
- Generics are specified as part of the class definition. We are able to show the parameter types that can be generalised by the class.

Syntax:

```
[public] class ClassName<Param0[,Param1..]>
```

Example:



Generics in Classes

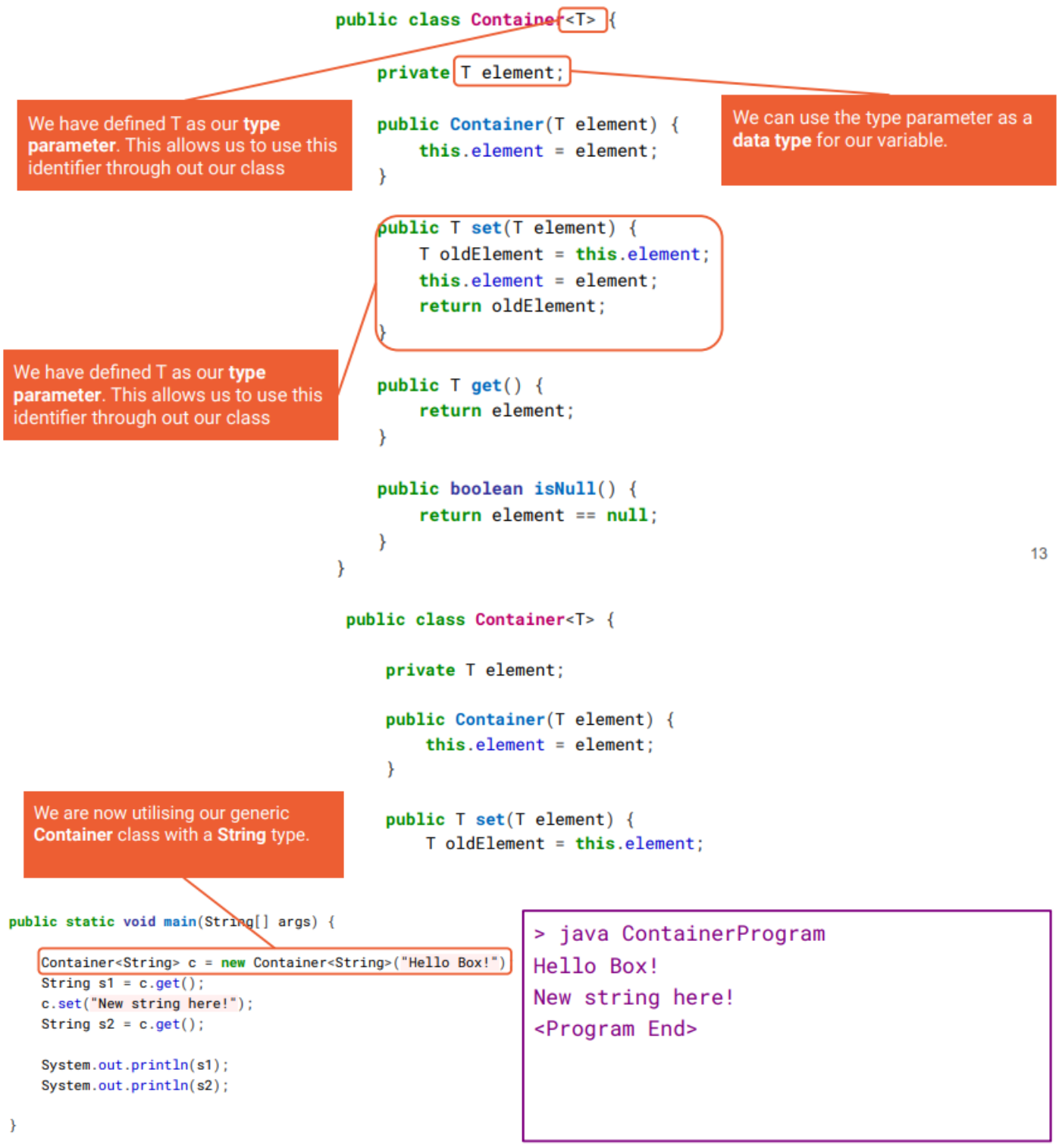
- We will use the generic identifier within our class so we can annotate methods and variables with it. This allow the method to be annotated with the generic variable.

Syntax:

```
[public] [static] T methodName()

[public] [static] void methodName(T parameter)

T variable

Type<T> variable
```

Generic Container

- In the following example, we will be writing a container that will store any type we want.




```
public class Container<T> {

    private T element;

    public Container(T element) {
        this.element = element;
    }

    public T set(T element) {
        T oldElement = this.element;
        this.element = element;
        return oldElement;
    }

    public T get() {
        return element;
    }

    public boolean isNull() {
        return element == null;
    }
}
```

```
public class Container<T> {

    private T element;

    public Container(T element) {
        this.element = element;
    }

    public T set(T element) {
        T oldElement = this.element;
```

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

    Container<String> c = new Container<String>("Hello Box!");
    String s1 = c.get();
    c.set("New string here!");
    String s2 = c.get();

    System.out.println(s1);
    System.out.println(s2);

}
```

- The following three lines allow you to infer what type is going to be used through the generic identifier.

Motivation

- Generics stem from the need to have type guarantees with usage. We have looked into checked and unchecked operations but it is best to understand what motivates the decision to have this degree of checking.

- Without generics, we would be required to perform casting between objects (if we were to store then as an Object type within the data structure), or we will need to duplicate the class multiple times for different types.

Generics Case 1

- Let's say we live in a world without generics. Let's go through two scenarios where we want to implement an ArrayList for every type we use in our program.
- It becomes apparent that we will be duplicating code for every data type, we would need to create:
    - ➔ IntArrayList
    - ➔ DoubleArrayList
    - ➔ FloatArrayList
    - ➔ StringArrayList

Generics Case 2

- The previous problem can be resolved through casting, which, however, causes some issues.
- All reference types inherit from object, hence we could treat all instances as Object.
- We effectively generalise all instances to an Object[] that will contain each type.

How Do We Know What Types Are Stored?

- The compiler wouldn't know what is stored in the ArrayList → it could be any kind of element, however, this isn't the only issue.

Let's use the container as an example:

```
public class Container {

    private Object element;

    public Container(Object element) {
        this.element = element;
    }

    public Object set(Object element) {
        Object oldElement = this.element;
        this.element = element;
        return oldElement;
    }

    public Object get() {
        return element;
    }

    public boolean isNull() {
        return element == null;
    }
}
```

Without generics we will need to use the **Object** to refer to the any **Reference** type instance.

So, looking at set, how would we know what type is being added?

It is worse with calling code as any type that we call will need to handle cast this to the correct type.

A bit of trivia: In dynamically typed languages, variables refer to a **Box** (similar to our **Container class**).

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

    Container<String> c = new Container<String>("Hello Box!");
    String s1 = c.get();
    c.set("New string here!");
    String s2 = c.get();

    System.out.println(s1);
    System.out.println(s2);

}
```

It is worse with calling code as any type that we call will need to handle cast this to the correct type.

- Removing this information will lead to the following error:

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

    Container c = new Container("Hello Box!");
    String s1 = c.get();
    c.set("New string here!");
    String s2 = c.get();

    System.out.println(s1);
    System.out.println(s2);

}
```

Since this is a **String** variable and the return type is **Object**. The compiler cannot guarantee type correctness here.

```
Container.java:26: error: incompatible types: Object
cannot be converted to String
      String s1 = c.get();
                  ^
Container.java:28: error: incompatible types: Object
cannot be converted to String
      String s2 = c.get();
                  ^
2 errors
```

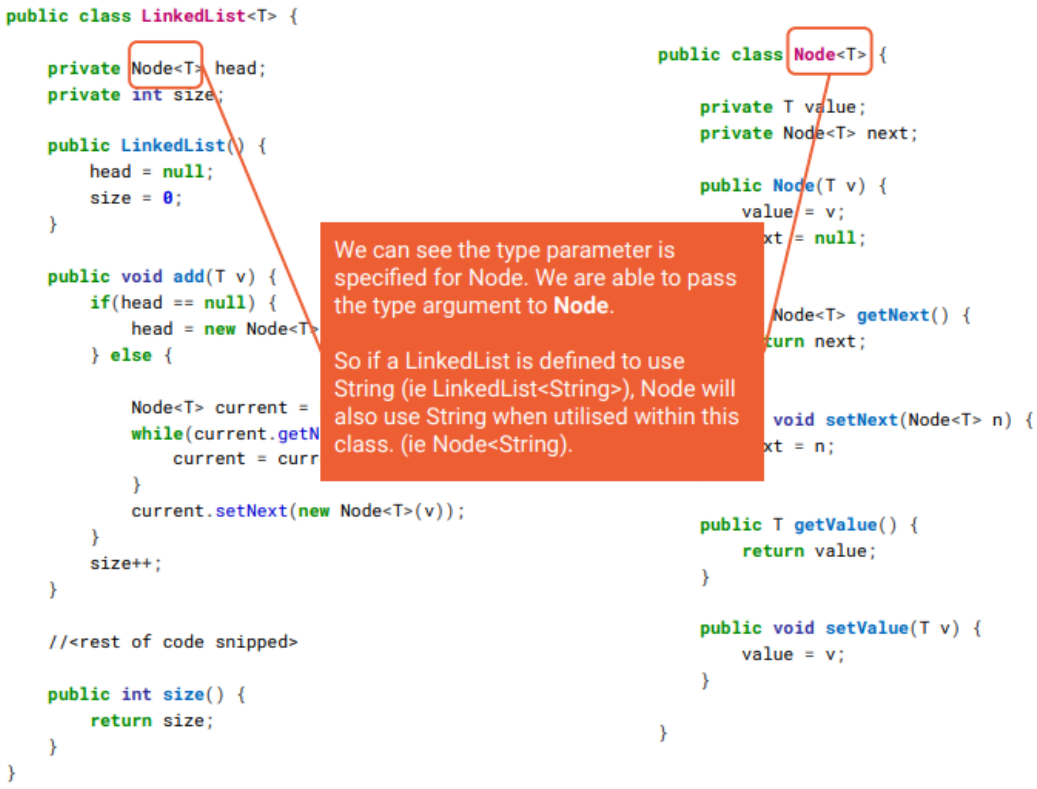
- We will need to cast the object to the correct type, which is a runtime check and can lead to unsafe assumptions.

    ➔ String s1 = (String) c.get();

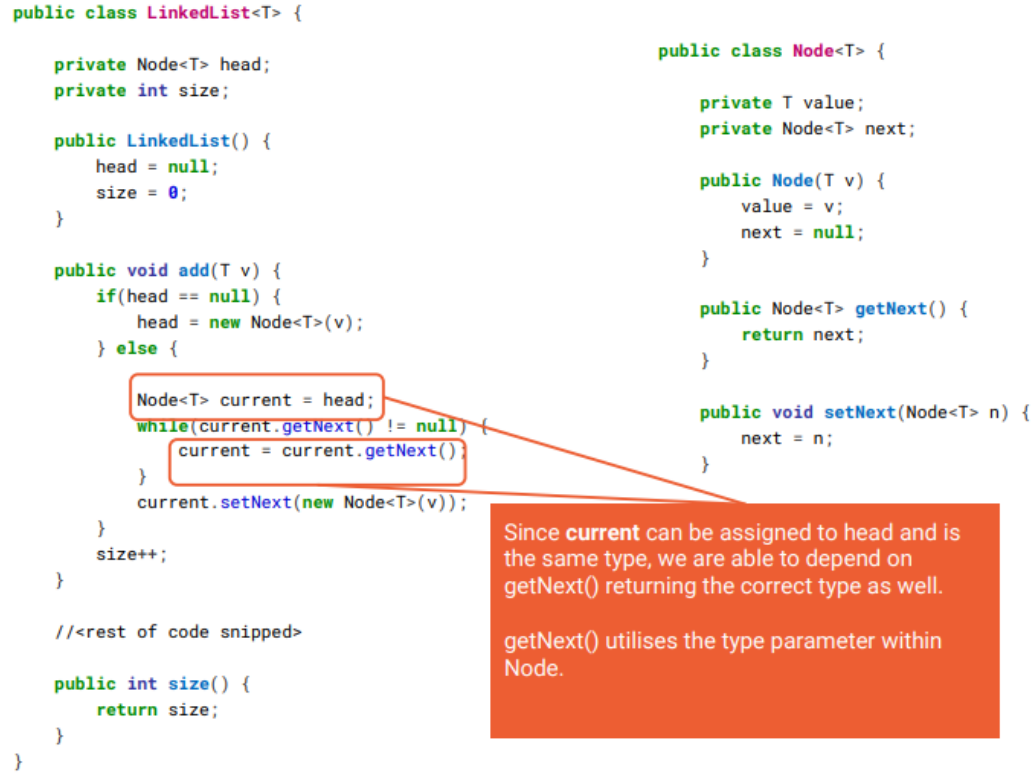
Usage of Generics in Data Structures

- Generics are typically used within collections as the operations and patterns involved do not differ based on the type that is used.
- A linked list that contains integers does not have different operations to a linked list that contains doubles.

- We typically provide a type argument when we use a collection. In this case, the <T> parameter is used throughout the class, instance attributes and even part of the method variable.



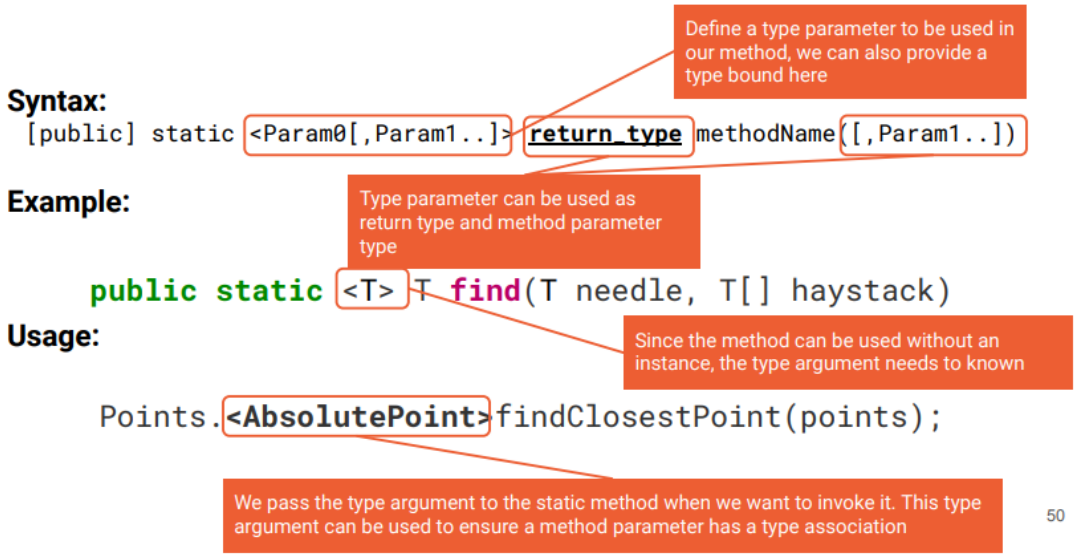
- The compiler's type system does a lot of work, verifying how we are utilising the variable even in the current context.
- Examining the usage: within the add method, we are utilising the .getNext() method of Node → the compiler knows what type we are using since we are using the type parameter on Node within LinkedList, forcing the generic type of both LinkedList and Node to be the same.



- As we saw when we remove the type argument in our collection types, the compiler is unable to check the type being used.
- Ultimately, this is a horrible idea → if we have this feature that allows us to get a guarantee from the compiler, we would want to utilise this, in effect knowing the binding and letting the compiler check for errors that we could make.

Generic Static Methods

- We can define a generic static method within our class that can operate on a set of data. Its syntax and usage is different than other instance methods as the type argument can be passed when called.



Type Erasure

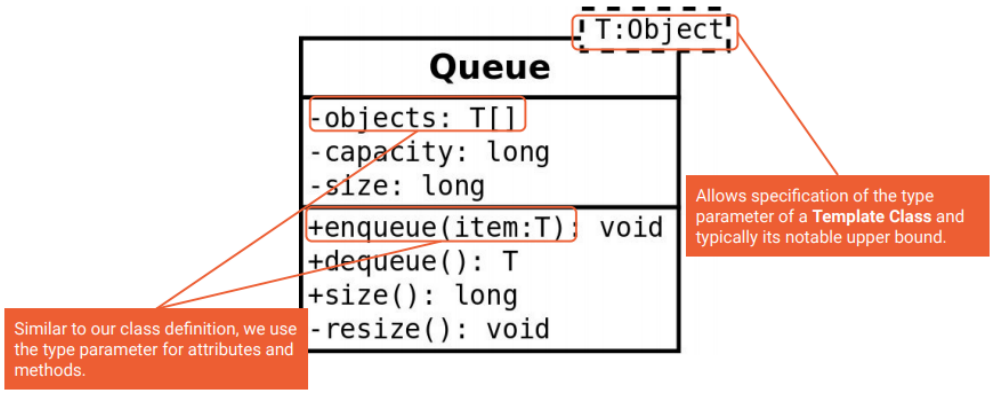
- Generics provide a compile-time guarantee and will prevent errors in our code prior to runtime. However, during runtime, type information is erased. You can consider the case where the compiler transforms all our get and add operations with casts.
- This data removal can make it very difficult to debug since there is a lack of information in regards to the type of data that is being stored.

Why is it Removed?

- By design, instead of duplicating the code for each, it will perform the checks at compile time and perform the casts accordingly. This is a compromise to maintain backward compatibility with existing java source code.

UML Template Class

- UML modelling language defines a class with generics as a Template Class. Within the visualisation, it will contain annotation of the type parameter.



extends with Type Parameters

- We can enforce constraints on what types can be used within the container. The rationale we have for this is that we may want to utilise explicit functionality of a super type.
- For example, we want to be able to create a class that contains any Shape class or any class that implements Drawable.
- Since the type parameter contains extra type data associated with it, we are able to guarantee access to methods from the super type with objects associated with the type parameter.

Type Parameters

- Looking back on our syntax with generics, we just simply specified an identifier for the type parameter. Now we can specify an upper bound type.

Syntax:

```
[public] class ClassName<Param0 [extends SuperType]>
```

Example:

```
public class ShoppingCart<T extends Item>
```

All types stored within **ShoppingCart** must extend from **Item**. We are then able to use methods defined in **Item**.

So let's break this down!

```java
public class Barrel<T extends Liquid> {

    private List<T> liquids;

    public Barrel() {
        liquids = new ArrayList<T>();
    }

    public void add(T liquid) {
        liquids.add(liquid);
    }

    public void outputVolume() {
        double maxLitres = 0.0;
        for(T e : liquids) {
            maxLitres += e.getLitres();
            System.out.println(e + ": " + e.getLitres() + "L");
        }
        System.out.println("Total: " + maxLitres + "L\n");

    }

}
```

```java
public abstract class Liquid {

    private double litres;

    public Liquid(double litres) {
        this.litres = litres;
    }

    public double getLitres() {
        return litres;
    }
}

public class Water extends Liquid {

    public Water(double litres) {
        super(litres);
    }

    public String toString() { return "Water";
}

}

public class Oil extends Liquid {

    public Oil(double litres) {
        super(litres);
    }

    public String toString() { return "Oil"; }
}
```


- Barrel<T extends Liquid> → as part of our class definition, we have included Liquid as our bounded type with the parameter. This infers that all types in this class must have a super type which is Liquid.
- private List<T> liquids; → this allows us to store any T type that is specified, this means that this barrel may only be used for Water, Oil, or Both, but this is defined by the user.
- public void outputVolume() → since we have a bounded type, we are able to infer that all types have a super type Liquid, therefore we are able to utilise the methods defined in liquid.

```java
    private List<T> liquids;

    public Barrel() {
        liquids = new ArrayLi
    }
```

```java
public double getLitres() {
    return litres;
}
```

As we can demonstrate here, we hve three separate instances that strictly contain each type or with the mixed one, any type that extends from Liquid.

```java
public static void main(String[] args) {
    Barrel<Water> waterBarrel = new Barrel<Water>();
    Barrel<Oil> oilBarrel = new Barrel<Oil>();
    Barrel<Liquid> mixedBarrel = new Barrel<Liquid>();

    waterBarrel.add(new Water(1.0));
    waterBarrel.add(new Water(2.0));

    waterBarrel.outputVolume();

    oilBarrel.add(new Oil(1.0));
    oilBarrel.add(new Oil(2.0));

    oilBarrel.outputVolume();

    mixedBarrel.add(new Oil(1.0));
    mixedBarrel.add(new Water(2.0));

    mixedBarrel.outputVolume();
}
```

```
> javac BarrelProgram
```

- If we were to add oil to a water barrel:

So let's break this down!

```
public class Barrel<T extends Liquid> {

    private List<T> liquids;

    public Barrel() {
        liquids = new ArrayLi
    }

public static void main(String[] args) {
    Barrel<Water> waterBarrel = new Barrel<Water>();
    Barrel<Oil> oilBarrel = new Barrel<Oil>();
    Barrel<Liquid> mixedBarrel = new Barrel<Liquid>();

    waterBarrel.add(new Water(1.0));
    waterBarrel.add(new Oil(2.0));

    waterBarrel.outputVolume();

    oilBarrel.add(new Oil(1.0));
    oilBarrel.add(new Oil(2.0));

    oilBarrel.outputVolume();

    mixedBarrel.add(new Oil(1.0));
    mixedBarrel.add(new Water(2.0));

    mixedBarrel.outputVolume();
}
```

When we attempt to compile the compiler will refuse to do this as the type safety is being violated here.

```
public abstract class Liquid {

    private double litres;

    public Liquid(double litres) {
        this.litres = litres;
    }

    public double getLitres() {
        return litres;
    }
```

```
> javac BarrelProgram
Liquid.java:21: error: incompatible types: Oil
cannot be converted to Water
    waterBarrel.add(new Oil(2.0));
                    ^
Note: Some messages have been simplified;
recompile with -Xdiags:verbose to get full output
1 error
```

Generic Arrays

- We have seen how we can use type parameters with single variables but how does it work with arrays?
- Not very well, as we will need to perform an unsafe operation to construct an array.

Let's see what happens if we were to **instantiate** a generic array?

```
public class DynamicArray<T, K extends T> {

    private T[] array;

    public DynamicArray() {
       array = new K[4];
    }

}
```

```
> javac DynamicArray
DynamicArray.java:8: error: generic array
creation
    array = new K[4];
            ^

1 errors
```

Drat! We are unable to instantiate a generic array.

- Let's take a look at this small program.

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

    String[] strings = {"One", "Two"};
    Object[] objects = strings;
    objects[0] = new Integer(1);

}
```

- String[] strings = {"One", "Two"}; → initialising an array of strings, completely valid operation of length two.
- Object[] objects = strings; → we are able to set objects to strings since arrays are covariant.
- Objects[0] = new Integer(1); → objects[0] is to be assigned an Integer type. Since Integer is a subtype of Object, this operation is considered valid – HOWEVER, since the original type is String[] , we have violated a type constraint at runtime.

```
> javac BrokenArrays.java
> java BrokenArrays
Exception in thread "main" java.lang.ArrayStoreException:
java.lang.Integer
    at BrokenArrays.main(BrokenArrays.java:7)
```

- Arrays are covariant, Generics are NOT!

With the following program which looks functionally similar to the Array version, we are unable to break this type constraint.

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

    List<String> strings = new ArrayList<String>();
    List<Object> objects = strings;
    objects[0] = new Integer(1); //Don't have to consider this

}
```

Specifically here, because **generics are invariant** we are unable to perform a similar operation to arrays, this will result in a compilation error.

Let's dive into building a **DynamicArray** with generics.

```
public class DynamicArray<T> {

    private T[] array;
    private int size;

    public DynamicArray() {
        array = new T[4];
        size = 0;
    }

    private void resize() {
        T[] temp = new T[array.length*2];
        for(int i = 0; i < array.length; i++) {
            temp[i] = array[i];
        }
        array = temp;
    }

    public void add(T v) {
        if(size >= array.length) {
            resize();
        }
        array[size] = v;
        size++;
    }
    //<code snipped>

}
```

There is a difference between declaring a generic array and instantiating a generic array.

Simply, this is because Arrays in java can break type safety where generics are aimed at enforcing type safety.

33

- array = new T[4]; → we need to do something unsafe now – we will need to instantiate an Object[] to use with array and cost: array = new(T[]) Object[4];

Iterators

- We have seen examples of the for-each loop. We will be looking into how we are able to implement the same behaviour on our own data structures.
- An iterator is an object that allows reading through a collection. It maintains state within the collection and where to go next.
- Prior to Java 5, the language did not have a language construct involving for-each (or enhanced for loop).
- However, iterators existed prior and therefore there exists a pattern that the for-each loop just translates into.

```
ArrayList<String> list = new ArrayList<String>();
for(String s : list) {
    System.out.println(s);
}
```

If we were to grab an item outside of the for-each loop, we would need to use **get()**. **How is the for-each loop doing this?**

But we need to consider the equivalence of this operation.

Here is our translation of a **for-each** loop, it is returning an iterator, using **hasNext()** and **next()** methods.

```
Iterator<String> iterator = list.iterator();
while(iterator.hasNext()) {
    String s = iterator.next(); //Returns element and moves it
    System.out.println(s);
}
```

We have access to an iterator object which in turn is used within a while loop. It will check that there is an element it can access next before iterating.

But this is Java, these methods must exist somewhere!

We can see the iterator has access to both hasNext() and next(), hasNext() checks, next() will return the next element in the collection.

- The iterable interface primarily declares an iterator() method to be defined by the collection type. The return iterator will be an object that can be utilised in a for-each loop.
- The iterator returned typically reflects the type that is utilised within the collection type. As per the language specs, the compiler will check if the collection has implemented iterable interface.

Method Summary

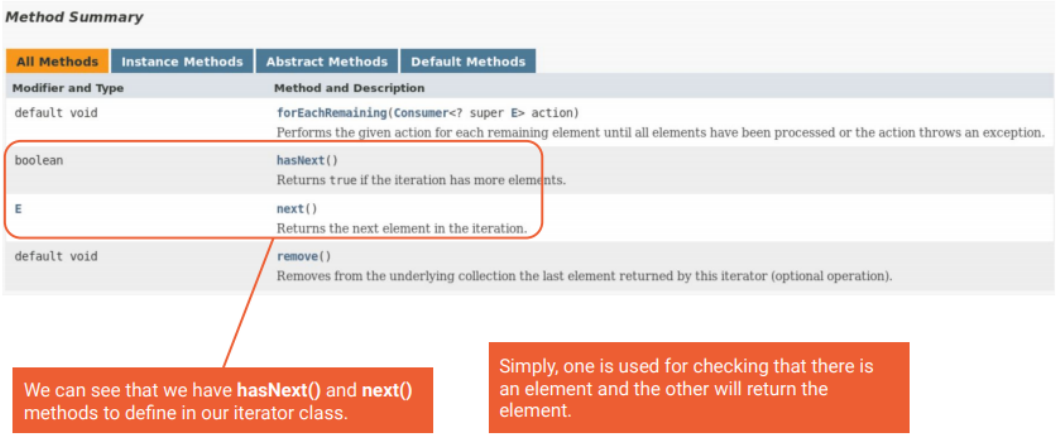
All Methods	Instance Methods	Abstract Methods	Default Methods
Modifier and Type		**Method and Description**	
default void		`forEach(Consumer<? super T> action)` Performs the given action for each element of the `Iterable` until all elements have been processed or the action throws an exception.	
`Iterator<T>`		`iterator()` Returns an iterator over elements of type `T`.	
default `Spliterator<T>`		`spliterator()` Creates a `Spliterator` over the elements described by this `Iterable`.	

We can see that it requires implementing a method called **iterator()** which will return **Iterator<T>** object.

Considering any class can create an iterator we will need to specifically mark types with **Iterable** for them to work with a for-each loop.

- It requires another type!

We can check out the iterator type from the java documentation and understand what methods compose an **iterator.**



Why Would This Be Preferred Over a For-Loop and Indexes?

- The class definition line specifies that this LinkedList will implement Iterable.

```
public LinkedList<T> implements Iterable<T> {

    private Node<T> head;
    private int size;

    public LinkedList() {
        head = null;
        size = 0;
    }

    public Iterator<T> iterator() {
        return ?;
    }

    public void add(T v) {
        if(head == null) {
            head = new Node<T>(v);
        } else {
            Node<T> current = head;
            while(current.getNext() != null) {
                current = current.getNext();
            }
            current.setNext(new Node<T>(v));
        }
        size++;
    }
    //<rest of code snipped>
    public int size() {
        return size;
    }
}
```

As part of the interface, we are required to implement **iterator()** method, however what do we return here?

We will create our own iterator that we will use in a for-each loop.

```
class LinkedListIterator<T> implements Iterator<T> {

    private Node<T> cursor;

    public LinkedListIterator(Node<T> head) {
        cursor = head;
    }

    public boolean hasNext() {
        return cursor != null;
    }

    public T next() {
        T element = cursor.getValue();
        cursor = cursor.getNext();

        return element;
    }

}
```

54

- We also contain a variable called cursor which will allow us to move through the collection.
- We also define our own hasNext() method.
- We write next() to return the next value whilst changing the cursor.

```
public Iterator<T> iterator() {
    return new LinkedListIterator(head);
}
```

- We update our iterator() method to return a LinkedListIterator object.

INFO1113 Week 8 Lecture – Exceptions, Enums, and Testing

Exceptions

- Exceptions can be considered an except state within our program.
- Exceptions evolve from the state of the machine or program execution being considered invalid.
    → Dividing by 0
    → Accessing memory that doesn't belong to your process
    → Dereferencing a null pointer
    → Unable to parse text
    → Out of memory
- There are a number of aspects we need to consider with the usage of exceptions.

Types of Exceptions

- There are different types of except states and severity, and different reasons to choose them.

Checked Exception

- This ensures you have handled the exception at compile time, it identifies a state that the programmer must handle.
- If we know the state can be violated and it is a common occurrence, an exception should be generated.

Runtime Exception (Unchecked Exception)

- Is a state that should occur, and you cannot handle nicely prior, unless you are expecting the code to raise an exception.
- If an error can occur and it is unreasonable for the caller to attempt to prevent the catching of the exception, then it can be a runtime exception → these are rare instances.

Error (State That Cannot Be Handled)

- Errors in Java can be handled by a try-catch block (but it is considered bad practice to catch them) and typically invoked when the state of the program is considered unrecoverable.
- Should be chosen if the state is unrecoverable and therefore should crash.

When Do We Use Exceptions?

- We need to consider a few aspects within our system e.g. when we write a function, we may need to make a few assumptions.
    → Using a floating-point variable, although the range of values should be between 0.0 and 1.0.
    → Only positive values accepted.
    → If the system is in an invalid state (order is in the state Paid but no payment has occurred).

Could Every Method Require an Exception?

- Yes, however exceptions and errors should be thrown when the precondition of the method has been violated.
- There are performance costs associated with throwing exceptions that involve stack unwinding and stack trace construction.

- We make assumptions with our applications and we want to ensure that these assumptions hold true by constructing a constraint that will notify the programmer of the error when violated.
- We need to also consider when use an exception and when we check.
- Exceptions are not a replacement for if statements!

Pre-Condition

- A precondition is input that must be written within its bounds for it to execute correctly.
- For example, if the method expects only a positive integer, any negative integer breaks the constraint afforded by the method.
- We may want to invoke an exception NegativeIntegerException, InvalidIntegerException, or something more specific to the problem domain.

Exceptions (cont.)

- Exception classes do not differ from any other classes besides extending from either Exception, RuntimeException and Error.

Syntax:

```
[public] class ExceptionName extends Exception

[public] class ExceptionName extends RuntimeException

[public] class ErrorName extends Error
```

Checked Exception

```
public static void imGonnaCrash() throws Exception {
    throw new Exception("Definitely crashing!");
}
```

Where we throw the exception, typically this is in some kind of if statement.

Since the method can throw a **checked exception** we are required it to handle it when we call it.

- Within the main method, we cannot proceed with:

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

- We are forced to catch it by the compiler:

```
public static void main(String[] args) {
    try {
        imGonnaCrash();
    } catch(Exception e) {
       e.printStackTrace();
    }
}
```

- HOWEVER, consider:

```
public static void imGonnaCrash() {
    throw new RuntimeException("Definitely crashing!");
}
```

Where the compiler will **not** force the programmer to handle a **RuntimeException.**

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

Example

```
public class Monitor {

    private double refreshRate;
    public final double MAX_REFRESH_RATE;

    public Monitor(double defaultRate, double max) {
        MAX_REFRESH_RATE = max;
        refreshRate = defaultRate;
    }

    public double setRefreshRate(double hz){
        refreshRate = hz;
        return refreshRate;
    }

}
```

In the following problem we are designing a system to set the **refresh rate** on a **monitor.**

We have implemented a simple method to set the refresh rate of the monitor object.

As part of this problem, the refresh rate should never be above **MAX_REFRESH_RATE**.

However we can see that in the current implementation we can easily **break** this rule.

This is where the pre-condition is violated and our method does nothing about it.

```
class InvalidRefreshRateException extends Exception {
    public InvalidRefreshRateException() {
        super("Unsupported refresh rate value");
    }
}
```

This is where we would implement an exception to show where the pre-condition has been violated.

- Following this, we will mark the method to throw an InvalidRefreshRateException.

```java
class InvalidRefreshRateException extends Exception {
    public InvalidRefreshRateException() {
        super("Unsupported refresh rate value");
    }
}

public class Monitor {

    private double refreshRate;
    public final double MAX_REFRESH_RATE;

    public Monitor(double defaultRate, double max) {
        MAX_REFRESH_RATE = max;
        refreshRate = defaultRate;
    }


    public double setRefreshRate(double hz) throws InvalidRefreshRateException {
        if(hz < 0 || hz > MAX_REFRESH_RATE) {
            throw new InvalidRefreshRateException();
        } else {
            refreshRate = hz;
        }
        return refreshRate;
    }

}
```

We add the logic to check that **refreshRate** can never be < 0 or > **MAX_REFRESH_RATE**.

Enums

- The java language provides a construct for enumerated types.
- Enums are a set of defined instances of the same type. An enum within java allows a finite set of instance to be constructed.
- We are unable to create unique instances of an enum type (we cannot use the new keyword).
- They are appropriate when we may run into problems where the number of instances are finite or manageable within a sequence of instances.
    - → A deck of playing cards.
    - → Telephone state (busy, offline, waiting, dialing)
    - → Laptop state
    - → Days of the week
    - → Months of a year
    - → Direction
- Enums are defined similar to a class but there are two variants of the construction. A C-like and a java-like construction.

Syntax:

```
[public] enum EnumName
```

Example (C-Like):

```java
public enum Suit {
  Hearts, //0
  Diamonds, //1
  Spades, //2
  Clubs, //3
}
```

Each instance has an ordinal number within the set. This also allows us to iterator through them

- Each instance of suit is labelled and can be referred to using the enum identifier.

Syntax:

```
[public] enum EnumName
```

Example (Java-Like):

```
enum Suit {
    Hearts(2, "Red"),
    Diamonds(1, "Red"),
    Spades(3, "Black"),
    Clubs(0, "Black");

    private int number;
    private String colour;

    Suit(int n, String colour) {
        this.number = n;
        this.colour = colour;
    }
    public String getColour() {
        return this.colour;
    }
}
```

- At the beginning, we are able to initialise each other instance at the start of the enum, passing parameters to it.
- Properties are then defined within type, we can refer to these variables within our methods.
- Following that, we have specified a constructor to be used. Each instance can invoke this and setup its attributes.

```
enum LightColour {
    Red,
    Green,
    Yellow;

    public LightColour change() {
        if(this == Red) {
            return Green;
        } else if(this == Green) {
            return Yellow;
        } else if(this == Yellow) {
            return Red;
        }
    }
}

public class TrafficLight {

    private LightColour colour;

    public TrafficLight() {
        colour = LightColour.Red; //By default it is Red.
    }

    public LightColour change() {
        colour = colour.change();
        return colour
    }

}
```

We have moved the change() method logic to the enum type. This allows us to specify it within the type instead of outside.

- Enums don't differ all that much from classes are we are able to utilise most class features with them.
    - → Implement interfaces
    - → Abstract methods
    - → Constructor overloading

→ Method overloading

→ Modify variables within a globally accessible instance

- Since we can define abstract methods, we can force each instance to contain their own implementation.

```
enum LightColour {
    Red{ public LightColour change() { return Green; } },
    Green{ public LightColour change() { return Yellow; } },
    Yellow{ public LightColour change() { return Red; } };

    public abstract LightColour change();
}

public class TrafficLight {

    private LightColour colour;

    public TrafficLight() {
        colour = LightColour.Red; //By default it is Red.
    }

    public LightColour change() {
        colour = colour.change();
        return colour
    }

}
```

We do not need to check. Each instance has its own transition method that specifies its return type.

Testing – Assert Keyword

- Assert evaluates an expression and will throw an Assertion Error if the statement is false.

Syntax:

```
assert expression
```

Example:

```
assert list.size() > 0

assert list.size() == 0 && writtenFiles
```

- Since it throws an Error type, it will cause our application to crash.
- assert is a keyword that allows us to test the truthfulness of a method or variable → we are able to test preconditions, postconditions and anything in between.
- It is not a substitute for control flow → the feature highlights anything you deem incorrect in your application, which may be difficult to consider, seeing as most states within our program can be recoverable.

→ Preparing to write updates to an operating system or large block of software.

→ Failure to write a core file that is necessary to your application running correctly.

→ Checking that methods provide the correct result and modifications to objects.

Post-Conditions

- A post-condition is where any mutation or output from a method is considered to adhere to the requirements of the method → what the method promises to do.
- E.g. a method must return the sum of numbers in a list. Failing this results in the post-condition being false.

```java
import java.util.List;
import java.io.File;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.StandardCopyOption;

public class PackageInstaller {

   //<snipped>

    private void preCheck() {
        File f = new File(pathPrefix);
        assert f.exists();
        assert f.isDirectory();
        assert key != null;
        assert keyInput != null;
        assert key.verify(keyInput);
        assert noFiles > 0;
        assert files != null;
        assert files.size() > 0;
        assert files.size() == noFiles;
        assert noFilesWritten == 0;
    }

}
```

```java
//<rest continues here>
private void commit() {

        for(File file : files) {
            try {
            Files.copy(file.toPath(),
                (new File(pathPrefix + file.getName())).toPath(),
                StandardCopyOption.REPLACE_EXISTING);
                noFilesWritten++;
            } catch(IOException e) {}
        }
    }

    public void install() {
        preCheck();
        commit();
        postCheck();
        cleanup();
    }

    private void postCheck() {
        assert noFilesWritten > 0;
        assert noFilesWritten == files.size();
        for(File file : files) {
            assert new File(pathPrefix+file.getName()).exists();
        }
    }
```

- The main method that will be called by our installer object is the install() method. This has a simple list of instructions to carry out.
- The first method being the precheck() method that will need to verify if all dependencies for the installation are satisfied.
- Each assert potentially will prevent the installer from progressing if it fails the check.
- If a precheck passes, then move to writing the files to the specified directory → commit().
- We will also run checks after writing the files to ensure that they have been written → we will need to check that all files were written as the commit method can skip files if an exception occurs.

Junit

- Although the compiler performs quite a number of checks for us to ensure we are using types correctly, it doesn't ensure that our program logic is infallible.
- When building any meaningful software project, you will need to formulate a mechanism of testing that the software complies with the requirements.
- A common testing framework in the Java ecosystem is Junit → simple framework that allows us to mark methods as tests.

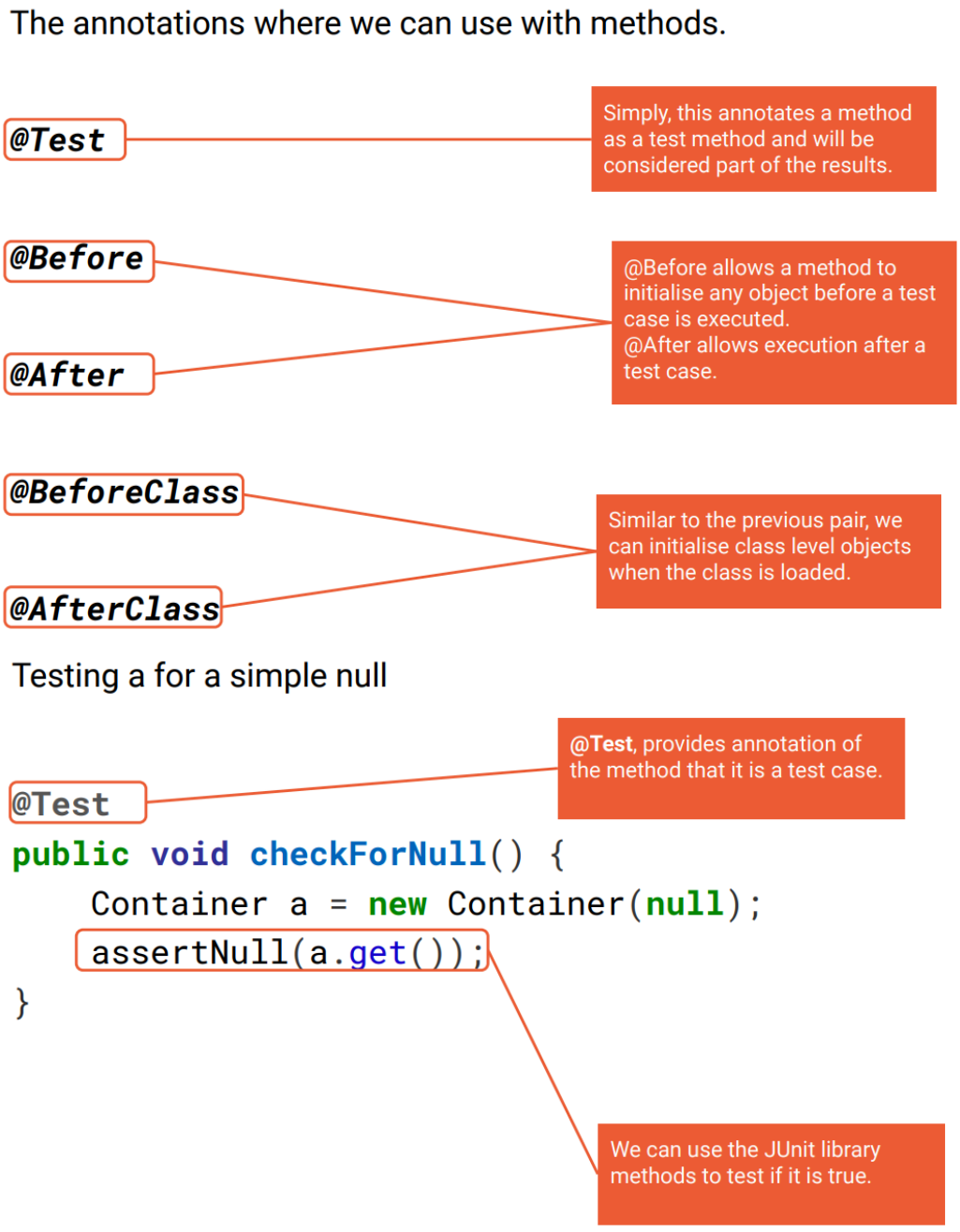
Types of Testing

- White Box Testing → this is typically where we employ some unit testing software, to help analyse the internals of the system and test them independently.
- Black Box Testing → user centric testing, without knowledge of the internals, input is given and compared to match the output of the program.
- Regression Testing → when the system has been modified and the changes may result in a failure of a previous successful test case.

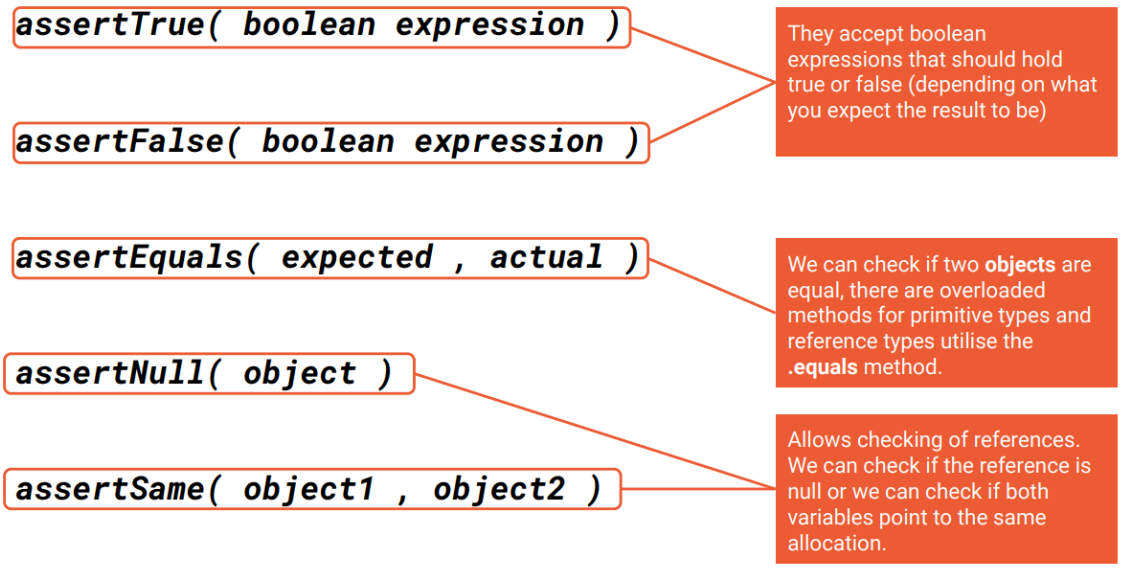
- Integration Testing → when developing individual components, we want to integrate it into the whole system and check to see if it works.

Running JUnit

- To set up Junit, you need to acquire junit.jar and hamcrest.jar files that are used to run JUnit.
- Within the java ecosystem .jar files (Java Archive) are a collection of a .class files that we can import into our own application. It exposes a whole new set of methods.
- The Java platform is typically geared towards IDEs and sometimes it can be troublesome to strictly live in a command line world with Java.
- This isn't true for a lot of environments which have a different set of tooling that affords interaction with a command line interface and an IDE.
- Within JUnit we have access to a variety of annotations that allow us to determine an order of execution for some of our methods and also sort test execution if so needed.
- However, we should not need to order test cases → we may however need to create a preparation method.

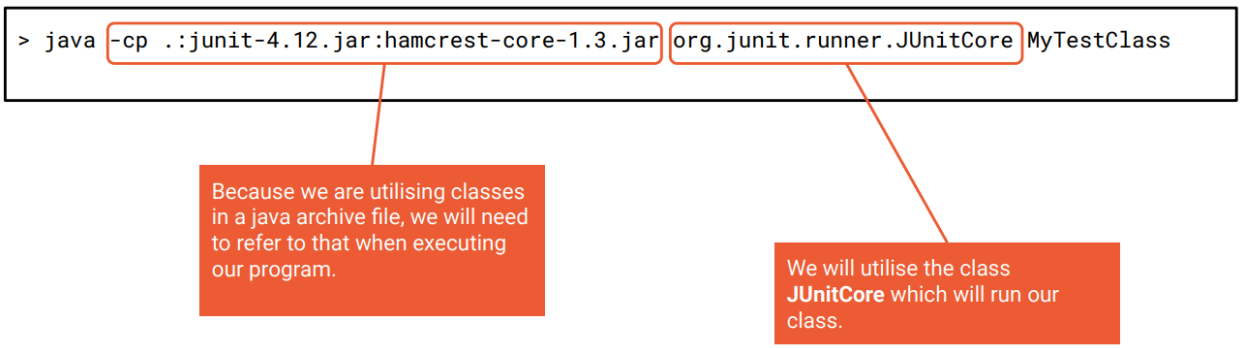


Our assert methods we have available within our JUnit.



- Once we have constructed out test case, we will need to compile it with the junit and hamcrest archives.

- To execute a JUnit class, we need to run the program differently from before.



INFO1113 Week 9 Lecture – Recursion

- Recursion is a technique within computer science that allows calling a function within itself.
- Recursive functions are aligned with recursive sequences or series', where the output of a function is dependent on the output of the same function with a change of input.
- Problems can often be represented easier with recursion → however, we are able to translate any recursive function to an iterative counterpart.

Parts of a Recursive Function

- A base case (or many base cases) where the function terminates.
- Recursive case (or many recursive cases) which will converge to a base case.

Drawbacks from Recursion

- The java programming model does not allow for infinite recursion.
- Inefficient with memory.
- Potentially more computationally demanding due to the overhead caused by method calls.

Recursion Example

```
public class Fibonacci {

    public int[] generateSequence(int n) {

        if(n < 0) {
            return new int[0];
        } else if(n == 0) {
            return new int[] {0};
        } else if(n == 1) {
            return new int[] {0, 1};
        } else {
            int[] f1 = generateSequence(n-1);
            int[] f2 = generateSequence(n-2);
            int[] newF = new int[f1.length+1];
            newF[newF.length-1] = f1[f1.length-1] + f2[f2.length-1];
            for(int i = 0; i < f1.length; i++) {
                newF[i] = f1[i];
            }
            return newF;
        }
    }
}
```

- generateSequence is a recursive method that takes in an integer, with base cases n < 0, n == 0 and n == 1 (these simply return an element to the caller).
- In this instance, the recursive method calls the same method with a change of input → N-1, N-2.

Reasons to Use Recursion

- Recursive methods can be simpler to read and write.

- Immediately writing an iterative method where there is an established recursive method can be considered a premature optimisation.
- Other systems that afford recursion do not suffer the same limitations as Java.

Recursion with OOP

- To extend from regular recursion, we are able to utilise objects with recursion.
- We are able to write the recursive method for class instances e.g. within an instance object of type FamilyMember, we could invoke, getChildren() which may call the same method on all FamilyMember ojbects that are children of the callee.
- This kind of recursion is common with linked data structures such as:
    - → Trees
    - → Linked List
    - → Graphs
    - → Stacks
    - → Queues
    - → Heaps
- Not only are we writing recursive methods, we are writing them for instances and generalising the usage for all instances.
- For any given instance, we can apply this method, but we are able to extend this with the use of polymorphism.

```
class FamilyMember {

    private String name;
    private int age;
    List<FamilyMember> children;

    public FamilyMember(String name, int age) {
        this.name = name;
        this.age = age;
        children = new ArrayList<FamilyMember>();
    }

    // Snipped

    public List<FamilyMember> getAllParents() {
        List<FamilyMember> parents = new ArrayList<FamilyMember>();
        if(this.getChildren().size() > 0) {
            parents.add(this);
            for(int i = 0; i < this.getChildren().size(); i++) {
                parents.addAll(this.getChildren().get(i).getAllParents());
            }
        }
        return parents;
    }
}
```

- We contain a list of children, or in a more abstract sense, links.
- Each FamilyMember contains a list of children, we retrieving a list of parents from a FamilyMember, we need to check each link if they are also a parent.
- Since each child is a FamilyMember type, we are able to call the method getAllParents() recursively, since the method adds all the elements to a list we are able to add it to the caller's list.

Memoization

- Memoization is a technique for storing the results of a computation → caching.
- We keep the result as we may want to reuse it later → e.g. if we had a website that computes a simple page, since the page doesn't differ between each user accessing, we could keep the result and send it every time it is asked.
- Recursive calls can be computationally expensive and if we are repeatedly calling dependent values or the same values, it makes sense to keep a record of that → we are maintaining a copy of the answer because other computations depend on it.

```
public class FibonacciCache {
    private Map<Integer, int[]> cache;
    public FibonacciCache() {
        cache = new TreeMap<Integer, int[]>();
    }
    public int[] generateSequence(int n) {
        if(cache.containsKey(n)) {
            return cache.get(n);
        } else {
            if(n < 0) {
                return new int[0];
            } else if(n == 0) {
                return new int[] {0};
            } else if(n == 1) {
                return new int[] {0, 1};
            } else {
                int[] f1 = generateSequence(n-1);
                int[] f2 = generateSequence(n-2);
                int[] newF = new int[f1.length+1];
                newF[newF.length-1] = f1[f1.length-1] + f2[f2.length-1];
                for(int i = 0; i < f1.length; i++) {
                    newF[i] = f1[i];
                }
                cache.put(n, newF);
                return newF;
            }
        }
    }
}
```

- We introduced a collection that will hold our answers. For convenience, we are using a Map.
- It is then constructed with an Integer as a key (the nth Fibonacci sequence) int[] as the value in the FibonacciCache() method.
- We have added a check at the start of generateSequence() to see if we have already computed this answer before → if we have, simply return it!