# Week 1

## Lecture 2: Primitive Types, Variables, Scope

Primitive:

* Built into the language
* Cannot be decomposed into simpler components
* Not a class (not objects)
* (technical) Variables of this type hold the value, not a reference
* Examples
  + byte, short, int, long, float, double, boolean, char
* Every type has an associated wrapper class
* Operators
  + Assignment
    - int i = 5;
  + Equality comparison
    - if (i == 5) { … };
  + Relational operators ( <, >, <=, >= )
  + Boolean combinations ( && (and), || (or) )

Java strings are objects with special support

* Immutable

Type conversion:

* Java is **statically-typed** => convert before reassignment
* Implicit (widening) conversions
  + Without extra source code
  + Widening – byte to long, long to double, …
  + Precision may be lost, but the magnitude is preserved
* Explicit (narrowing) conversions
  + Can result in significant potentional info loss
  + Have to use *casting*
    - int i = 1025;  
      byte b = (byte)i  
      // b now has value 1
  + Examples
    - double to float
      * loss of precision
      * out-of-range values become infinite
      * some non-zero values may become zero
    - double/float to long/int/short/byte/char
      * round-to-zero
      * out-of-range becomes infinity
    - Integer type to “smaller” integer type (long to int, int to byte, …)
      * discard all but n lowest-order bits
      * in practice – value *mod* max
  + String primitive types
    - Using parseXXX method of primitive wrapper classes
      * int i = Integer.parseInt(“42”);
  + Primitive to String
    - Concatenate (+) with an existing String value (like “”)
    - Or String.valueOf(…)

Integer division:

* If both are integers (int, long, short, bytw, char), then **integer division**
  + 7/4 = 1
* If either is floating-point (float, double), then **floating point division**
  + 7.0/4.0 = 1.75

Variable:

* a container that stores info
* Statically typed:
  + Declaration – allocating memory for use
  + Initialisation – giving a value

Identifier names:

* Identifier – label for a named Java entity
  + Class, field, method, parameter, local variable
* The are rules and regulations
  + Rules: have to comply for the program to compile
    - Identifiers must begin with a letter or an underscore (“\_”)
    - Must be at least 1 character long
    - Other characters may be letters (Unicode), numbers, or underscores
  + Conventions: legible code
    - Classes start with a capital letter
    - Other identifiers start with a lowercase letter
    - Multi-word identifiers use “CamelCase”
    - Constant values are written in ALL CAPS

## Q&A

## Lecture 3: Control Structures, Methods

Control structure types:

* conditional execution (if, switch)
* iteration (for, while)

Block of statements is enclosed in curly braces {}

* A variable is in scope from the point of declaration to the end of the enclosing block

For loops:

* Contain three clauses
  + Initialisation
  + Termination condition check
    - Checked before loop is executed
  + Variable update
* Variable local to the loop

Loop control flow:

* break – break out of the loop
* continue – skip the rest of the block

If versus Switch:

* If evaluates any boolean expression
  + Switch evaluates an integer or String
* if conditionally executes one of two statements or blocks
  + switch conditionally executes one or more of a list of case blocks
* Use depends on readibility, complexity

Switch:

* uses cases
* default – “none of the above”
* fall-through – putting multiple cases for one executive block

Methods:

* Divide a problem (code) into manageable sub-problems (chunks)
* Removes repetition
  + Easier and faster to read
  + Faster to type
  + Easier to error check
* Parameters – receives values from code that called it
  + Declared types
* If method is called, it runs statements in order
* May return a value
  + return type needs to be specified
* Variables are local
  + But parameters cannot be changed globally
* Declaring:
  + 1. access modifier(s) (zero or more, public)
  + 2. return type (void if none)
  + 3. name (usually verb) (part of method signature)
  + 4. parameter list in brackets (part of method signature)
  + 5. an exception list
  + 6. the body (enclosed in curly braces)
* Calling:
  + use name
  + specify parameters
  + store result in a variable of the correct type (only if non-void)
* Overloading: having multiple methods with the same name but different parameter lists

# Week 2

## Lecture 4: Arrays, Non-Primitive Types

Array – a fixed length sequence of consecutive memory locations, indexed by an integer subscript

* Each item is called an element
* Each element is accessed by index
  + First index is 0
* Each array has:
  + A type – type of individual elements
  + A dimension – the dimensionality
* Declaring – type and dimension
* Initialising – reserving space and specifying length of the array
  + Only creates a reference to point to the array
  + Two options:
    - “new” and size/values
      * int[] values = new int[10]
      * Fills with 0 for numeric types
      * Fills with Null for other types
    - using an initialiser
  + Directly:
    - int[] values = { 10, 20, 30, };
    - Only do this if declaring and initialising in the same lines
* Access: use square brackets after variable with index
* Length cannot be changed
  + strings.length (without brackets)
* Iterating:
  + Standard *for* loop
  + More efficient: *for-each* loop
    - for (String fruit: fruits){  
       System.out.println(fruit);  
      }
* Manipulation with Arrays class
  + Arrays are weird objects
    - One field (length)
    - No methods except inherited from Object
  + java.util.Arrays provides useful methods:
    - Arrays.copyOfRange – copies subarray into a new array
    - Arrays.equals – compares
    - Arrays.fill – fills with a specific element
    - Arrays.sort – sorts in ascending order
    - Arrays.toString – nicely formatted version
* Scope:
  + Cannot change the memory location of an array, but can change its elements (one by one)

Non-primitive (composite) types in Java:

* Examples:
  + Strings
  + Arrays
  + Objects
* Properties:
  + Not always built into language
    - Strings and Arrays are
  + Can be decomposed into simpler components
    - Arrays are made up of elements
    - Strings made up of characters
    - Each state of object is represented as fields
  + Are classes
    - Class – a definition of a type, including state and behaviour
  + Variables of this type hold a reference, not the value
    - Comparing with “==” compares **memory locations**
  + Can be defined by programmer

## Lecture 5: Objects, Access Modifiers

### Objects

Characteristics of objects:

* State (properties)
* Behaviour (possible actions)

Usefulness of using objects:

* Modularity (easy to read, debug, work on with a team
* Info hiding
* Code reuse
* Pluggability and debugging ease

Classes vs objects:

* Classes are types
* Objects are instances of types
* An object is an instance of a general class of objects

Abstraction – A class is an abstract description of real-world entities

Encapsulation – Each *instance* of a class has data and behaviour associated with it

Class declaration:

* *class*
* Name
* Body
  + Fields (properties)
    - Store state that represents some attributes of the objects
    - An instance variable is accessible in all methods of a class
  + Methods (behaviour)
    - Represent behaviour that processes and transforms the object state
* Optional: access modifiers, superclass, interfaces)
* Optional: constructor
  + A method with the same name as the class
  + No return type (no *void*)
  + Sets up initial values for the data fields to initialise object state
    - Using *this* to refer to current object being created
  + If no constructor is specified, a default *no-args* constructor is automatically created
    - No arguments
    - Sets all fields to their default values (usually 0 or *null*)
* Optional: getters (accessors) and setters (mutators)
  + Give controlled access to private properties
* Optional: unique ID
  + Possible with a static variable
  + Static - changes for all

Types of variables:

* Local variables (declared in a method)
* Method parameters (in a method header)
* Member variables in a class (fields)

Initialising object instance:

* BankAccount mary = new BankAccount (“Mary”, 0)

Calling method – variable.method(params)

* mary.deposit(50)

### Access Modifiers

Visibility modifiers:

* Types:
  + public: same class, same package, any subclass, any class
  + protected: same class, same package, any subclass
  + (default): same class, same package
  + private: same class
* Used to limit the visibility of class members
  + Getters, setters used to give controlled access

Static is the opposite of instance

Static members

* Associated directly with the class itself, not with any object of that class
* Static field: only 1 variable, no matter how many objects of the class have been created (including 0)
* Static method: performs a general task for the class; can only access other static members

# Week 3

## Lecture 6: Inheritance

Classes can inherit state and behaviour (fields and methods) from other classes

* Subclass and superclass

In Java, a class can have exactly one superclass. If not specified, it’s Object.

* Single inheritance
* “[class] extends [another class]”

Subclasses can override superclass methods to provide specialised behaviour

Superclass’ field must be at least “protected” to be visible by subclasses

Subclasses’ methods:

* Use them directly (unless private)
* Override
  + Can override built-in methods (like toString) for easier legibility
* Hide a static method by writing new one with same signature
* Declare new ones

Polymorphism – whenever an instance of class A is expected, one can also use an instance of any subclass of A

* If a method is overridden in a subclass, Java will always use the most specific overridden version (even when the variable type is the superclass)
* Supported by virtual method invocation: method calls are dynamically dispatched based on the runtime type of the receiver object

Accessing superclass methods:

* Using “super” instead of “this”

## Lecture 7: Inheritance (details)

Constructors:

* can be multiple if they’re with different amount of parameters
* a constructor can call another constructor with “this([params])” inside it
* Default (“no-args”):
  + takes no parameters
  + sets all fields to default values (0 for numeric types, false for Boolean, null for non-primitive types)
  + inserts a call to the parent class’ no-arg constructor – super()

Constructors and Inheritance:

* Constructors from the super class can be invoked by the subclasses (but not inherited from them)

Access modifiers and Inheritance:

* private members cannot be inherited
* The subclass method can allow more, but not less, access than the superclass method

Instance and static methods:

* For instance methods, Java will always **execute the method in the subclass** (polymorphism), whatever the run-time type
* For static methods, which method gets chosen depends on how it is called (i.e. which class is used)

# Week 4

## Lecture 8: Final, Abstraction, Interfaces

Inheritance issues:

* Polymorphism allows using any instance of any subclass of a superclass
* If the superclass has a critical function, inheritance allows **subclass injection attacks**

Final:

* class – no subclasses
  + Useful for immutability (String/Double)
  + If all methods require *final*
* method – no overriding
  + Predictable behaviour
  + More reliable security
* field – unchangeable value
* parameter – unchangeable value inside a method
* variable – unchangeable value
* *static final* – constants
  + Usually written all caps
  + Ex: Math.E, Long.MAX\_VALUE

Abstraction:

* A superclass with methods requiring overriding
  + A class with abstract methods must be abstract
  + If a subclass does not implement all abstract methods, it is also abstract
* Cannot instantiate abstract classes
  + Only concrete (opposite of abstract) subclasses
* Ensures that all subclasses follow a certain API/structure
* Abstract methods have no body

Interfaces:

* Represent class relationships outside of main inheritance hierarchy, they are a contract that all implementing classes must honour
* Allow multiple inheritance of type
* Specify a public API
  + Implementation is irrelevant – method signatures only
* Methods do not have bodies
* Ex: “public class MyClass **implements** Interface1, Interface2 {}”
* Specifics:
  + Declared with interface keyword – “public interface TalkingCreature{}”
  + All (non-static) methods are implicitly “public abstract” (no need for explicit)
  + All fields are implicitly “public static final” (constants)
  + Has no instance-level fields or methods
    - Except default ones
    - Eliminates issues with multiple inheritance of state and implementation
* Implementation:
  + “implements” after “extends”
  + Provide definition to all methods declared in each interface (or declare class abstract)
    - Must be public methods
* Conflicts:
  + Cannot implement interfaces with methods with same name and params
  + Can implement interfaces with methods with identical signatures
* Using as a type:
  + Cannot instantiate (through new), but can use as data type (variables, method params, etc)
* (Default methods)
  + Allows implementation to be provided
  + Implementing classes
    - Don’t need to mention method (default behaviour inherited)
    - Can redeclare method (becomes abstract like a normal interface method)
    - Can redefine method (overriding)
  + Added to allow interfaces to be extended/upgraded

|  |  |
| --- | --- |
| **Interfaces** | **Abstract classes** |
| Can’t be instantiated | Can’t be instantiated |
| No constructor | Constructor(s) |
| All methods are public | Methods with any access modifier |
| Only constant fields | Constant and “normal” fields |
| Classes can implement multiple interfaces | Classes can have at most one parent class |

## Lecture 9: Exceptions, Style

### Exceptions

* Program crashes unless exception is caught
* Checked and Unchecked
  + Checked
    - Must be explicitly handled
      * Program will not compile otherwise
    - Generally indicate conditions that a well-written application should **anticipate and recover from**
    - Ex: FileNotFoundException
  + Unchecked
    - No need to be explicitly handles
      * Program still compiles and runs
    - Generally indicate **programming/logic bugs** that an application cannot reasonably recover from
    - Ex: ArrayIndexOutOfBoundsException
* Handling
  + Catching
    - Wrap a “try{}” block around code that might throw an Exception
    - Must be followed by 1+ “catch{}” blocks
      * First one with matching exception executed
      * “catch(Exception ex){}”
    - Optional “finally{}” block
      * Executed after entire rest of the try block
  + Passing on
    - Add “throws Exception” to method
    - Anyone calling that method must handle the exception (catching/passing on)
* Consists of
  + A message
  + A call stack – the sequence of method calls that ultimately resulted in the error
    - ex.printStackTrace()
      * Often has line numbers
      * Helpful for debugging
* Throwing
  + “throw new Exception (“Invalid input”);
  + Can throw at any point
  + String param indicated msg (ex.getMessage())
  + If checked exception thrown, must be added to the header of the method
    - “public String processInput (String input) throws Exception{…}”
* Advantages
  + Separating out error-handling code
    - Legible
  + Propagating errors up the call stack
  + Grouping error types
    - Since Exception is a class, it can be subclassed

### Coding Style

* Subjective (can turn into “bikeshedding”, cf. Law of Triviality)
* Conventions (Google, Apache commons, Spring)
* Declarations
  + Class/interface name starts with capital letter and be in PascalCase
  + Public class should be saved in file with same name
    - One public class per file
  + Variable/field names – descriptive, but not too long
    - schoolId
  + Methods start with verb and in camelCase
  + Static fields in ALL\_CAPS and with underscores
* Order of members in class/interface
  + Group related members together
  + Fields first, one per line, blank lines between groups
  + Constructors after fields
  + Methods after constructors
    - Logical order: accessors/mutators together, related methods together
    - Group all overloaded methods (same name) together
* Indentation/Spacing
  + Not required, but extremely important for readability
  + General rule:
    - Every new block indented one more unit
    - Many style guides insist on spaces (not tabs)
  + Line width – 70-120 characters
  + A good IDE will manage it for the programmer
  + Spacing between blocks of code:
    - Closely related lines together
    - Blank line between separate “thoughts”
* Commenting
  + Two types:
    - Implementation comments – particular aspect of code
      * Describing block of code (often in conjunction with blank line before)
      * Calling out/explaining non-obvious implementation decision
      * Not trivial
      * Above block of code
      * //
    - Documentation comments – describe code specification to outside reader
      * Java docs
        + Processes Java source files and generates HTML documentation
        + Reflects structure of source files
      * Javadoc comments
        + Surrounded by /\*\* … \*/
        + Refer to the class member that comes directly below them
        + Can contain HTML
        + Tags (with @)

# Week 5

## Lecture 10: Packages, Collections

Packages

* \* - related resources grouped together (usually classes)
* Reasons to use
  + Make it obvious that types are related
  + Easier to find types for a specific purpose
  + Type names won’t conflict from other packages
  + Types within package can have unrestricted access to one another
    - And restrict for types outside of package
* Creating
  + Choose a name
    - Rules are the same as identifiers
      * Can’t start with a digit
      * Can’t contain special characters/hyphens
      * Can’t contain reserved keywords (int, new, …)
      * Components separated by period
    - Conventions
      * All lower case
      * Built-in packages start with java. or javax.
      * Companies and organisation usually use their domain name, reversed
        + com.example.mypackage
        + uk.ac.glasgow.dcs.jp2
  + Put a package statement at the top of every source file for that package

package my.package.name;

* + Ensure all source files are in a directory corresponding to the package name
  + If no package statement, then all files are in the default package
* Accessing package members
  + To use a public type from outside the package, either
    - refer to it by its fully qualified name
      * Ex: java.util.Scanner stdin = new java.util.Scanner(System.in);
      * Works well for infrequent use
      * Code can easily become repetitive and hard to read
    - import the member itself
      * Ex: import java.util.Scanner; (at the top of the source file after package statement)
      * Ex (cont.): Scanner stdin = new Scanner(System.in)
      * Works well if you use a few members from a package
    - import the entire package
      * Ex: import java.util.\*; (at top, after package stmt)
      * Ex (cont.): Scanner stdin…;
      * Useful if you nee lots of members from the same package
      * Use is controversial (takes a lot of space
  + Note: java.lang is imported by default
    - Ex: String, Double, Exception
  + **NOT** in hierarchy
    - import java.awt.\* will not import any classes from java.awt.color
  + Corresponds to file directory
* Eclipse shortcuts
  + Ctrl-Space on (partial) class name
    - Tries to auto-complete package
  + Ctrl-Shift-O
    - Organises imports (removes unused, sorts rest)

Collections

* A standard set of built-in classes for representing and manipulating collections
* Each class groups related elements into a single unit
* Ex:
  + ArrayList – variable-length arrays
    - More flexible than built-in, but less efficient
    - Acts as a wrapper around an underlying array that grows and shrinks dynamically
    - ArrayList is a class => elements are added and removed by methods
    - Has a capacity (size of internal array) and a size (number of elements in the list)
      * Capacity is increased when necessary – purely internal
      * Size is increased/decreased as elements are added/removed and checked for operations
    - Creating
      * List<String> strings = new ArrayList<>(); (default initial capacity (10)
      * List<String> strings = new ArrayList<>(50); (explicit init capacity)
    - Adding
      * strings.add(“foo”) – adds last
      * strings.add(5, “foo”) – adds at index 5, shifts other elements
      * strings.set(5, “foo”) – sets at index 5
    - Accessing – get(ix)
    - Removing – remove(ix) or remove(el) (first occurrence)
    - Checking
      * contains()
      * indexOf()
      * lastIndexOf()
    - List vs ArrayList
      * List is the high-level Collection interface
        + Specifies methods
      * ArrayList is the specific type of List
        + Provides concrete implementation
        + Additional methods related to capacity
      * Use ArrayList
        + when initialising new variable
        + if you want to manipulate capacity
      * Use List at all other times (allows implementations to be swapped cleanly)
    - toString() looks nice
  + HashSet – a group of unique elements
  + Stack – a list with LIFO semantics
  + HashMap – a dictionary
* Base class: java.util.Collection
* Methods: add(), remove(), contains(), size(), toArray()
* Advantages
  + Reduces programming effort with pre-written data structures and algorithms
  + Increases performance with high-performance implementations
    - Interchangeable implementations – can switch to tune performance
  + Provides interoperability by allowing Collections to be passed back and forth
  + Reduces effort to learn new APIs bc of common interface
  + Reduces effort to design APIs by giving design specifications
  + Fosters software reuse by providing standard interface

## Lecture 11: Collections+ (Sets, Maps), Objects, Comparable

Generic types:

* Collection classes are type-parameterised
  + The type specified in angle brackets (<>) after the name specifies the type of the elements stored in that collection
  + No type specified -> Object
* (Polymorphism – subclasses of specified type will also be accepted)
* Added to Java in Java 1.5 (2004)
* Reasons to use
  + Compile-time error checking
  + Iteration can be much cleaner

Primitive types and generics

* The <type> generic parameter needs to be a class
  + Primitive types will not work
  + Use wrapper classes (Int, Long, etc)
* Boxing and unboxing
  + Java automatically converts between wrapper classes and primitive types

Sets

* Interface: java.util.Set
  + Concrete implementations: HashSet, TreeSet, LinkedHashSet
* Differences with List
  + Cannot contain duplicate elements
    - add() enforces this – returns t/f whether element was already in set
  + 2 sets are equal if they contain the same elements, regardless of implementation
* Uses
  + Finding unique values

Maps

* Interface: java.util.Map
  + Concrete implementations: HashMap, TreeMap, LinkedHashMap
* Provides a mapping from keys to values
  + Cannot contain duplicate keys
  + Each key maps to exactly one value
* Methods
  + get(key) – returns value (null if no value)
  + put(key, value)
* Uses
  + Count word frequency

Methods of Objects:

* boolean equals(Object obj)
  + reflexive (x equals x)
  + symmetric (x equals y and y equals x)
  + transitive (x equals y, y equals z => x equals z)
  + consistent
  + Default
    - returns true if and only if x and y refer to the same object
    - Gives the correct result for primitive types
    - Does not check if objects are equivalent (same content)
* public int hashCode()
  + returns hash code value (an int)
  + consistent (during 1 execution of an app)
  + Default
    - returns basically the memory address of the object

Comparable:

* Built-in interface that declares how objects are compared to one another for sorting
  + If a class implements Comparable, then lists of that type can be sorted
* Defines compareTo(), which returns
  + < 0 if this object is less than the other one
  + > 0 if this object is greater than the other one
  + = 0 if this object is equal to the other one
  + package java.lang;  
    public interface Comparable<T>{  
     public int compareTo(T o);  
    }
* Built-in uses
  + Collections.sort()
  + Arrays.sort()
  + SortedSet / SortedMap
  + Useful library classes
    - String
    - Long/Integer/Character/etc
    - Date
    - File

equals() and compareTo():

* Docs for Comparable say: strongly recommended, but not strictly required that
* (x.compareTo(y) == 0) == (x.equals(y))

# Week 6

## Lecture 12: File I/O (java.nio)

Package – java.io (old) or java.nio (new since 1.4, revised in 1.7)

Terms:

* Path – identifies a file through its location in the file system, beginning from the root node
* Delimiter – separates directory names
  + Windows \
  + Elsewhere: /

java.nio.file.Path:

* Represents a path in the file system
  + File name
  + List of directories to reach it
* Used to examine, locate, manipulate files
* Corresponds to the underlying file system – system dependent
* Corresponding file does not need to exist, one can manipulate a Path however one wants
  + Methods in “Files” class to deal with actual files

Relative vs absolute paths

* **Absolute** path always contains the **root** element and the complete directory
* **Relative** path needs to be combined with another path to access the file
* Path objects can be absolute or relative, depending if they begin at the root or not

Operations on Path

* Syntactic operations: operate on path itself, don’t touch file system
* Creating
  + Paths.get(“C:\\Users\\...”)
* Retrieving
  + Path stores name elements as a sequence
  + Different methods:
    - toString()
    - getFileName()
    - getName(index)
    - getNameCount() (no. of elements)
    - subpath(ind, ind)
    - getParent() – everything except filename
    - getRoot()
* Processing
  + Remove redundancies ( normalize() )
    - Cleans up path without checking file system
  + Converting
    - toAbsolutePath(): prepends current working directory
    - toRealPath(): returns the real path of an existing
      * converts relative path to absolute path
      * resolves symbolic links (if “true” is passed)
      * removes redundant elements
      * throws an exception if file does not exist (FileNotFoundException) or cannot be accessed (IOException)

Notes on Path:

* Implements Comparable
* Iterable
* equals(), beginsWith(), endsWith()

java.nio.file.Files

* Set of static methods for reading, writing, and manipulating files and directories
* Methods work on Path instances
* Most methods throw and IOException on I/O failure
* Checking
  + Verify existence
    - Files.exists(Path), Files.notExists(Path)
  + Check accessibility
    - Files.isReadable(Path), Files.isWritable(Path), Files.isExecutable(Path)
  + Whether two paths point to same file
    - Files.isSameFile(Path, Path)
* Deleting – Files.delete(Path), Files.deleteIfExists(Path)
* Copying – Files.copy (Path, Path)
* Moving – Files.move(Path, Path)
* Creating
  + Files.createFile(Path)
    - Throws exception if file already exists or parent dir doesn’t
  + Files.createDirectory(Path)
    - Throws exception if dir already exists or parent dir doesn’t
  + Files.createDirectories(Path)
    - Creates all necessary dirs.
* Listing
  + Path dir = …;  
    DirectoryStream<Path> stream = Files.newDirectoryStream(dir)  
    for (Path file : stream){  
     System.out.println(file.getFileName());  
    }
* Reading (text files)
  + Path path = …;  
    String content = Files.readString(path)
  + Path path = …;  
    List<String> content = Files.readAllLines(path); (one line at a time)
  + Path path = …;  
    BufferedReader br = Files.newBufferedReader(path); (for more control)
* Writing (text files)
  + Path path = …;  
    PrintWriter pw = new PrintWriter(Files.newBufferedWriter(path));  
    // pw.println()

## Lecture 13: GUI (Swing)

Java GUI toolkits: AWT, Swing, JavaFX

* AWT (“Abstract Windowing Toolkit”) – since the very beginning (1996)
  + First Java GUI toolkit: set of “heavyweight” classes using native GUI widgets
  + Still included in Java, not widely used
* Swing – since Java 2 (1998)
  + Cross-platform “lightweight” GUI components written entirely in Java
  + More powerful and flexible components than AWT
  + Still included in Java, widely used
* JavaFX (introduced in 2007, bundled with Java 8: 2014)
  + Un-bundled again as of Java 11; now separate open-source at <https://openjfx.io>
  + Somewhat widely used – seen as more flexible than Swing, but never got traction, now HTML5 appears to be taking over

Classic Model-View-Controller (MVC)

* Model
  + … updates …
* View
  + … is seen by …
* User
  + … uses …
* Controller
  + … manipulates … (Model)

Swing: modified MVC

* View and Controller together in UI delegate

javax.swing.JFrame

* Represents a top level window
* Operations
  + setSize (width, height)
  + setLocationByPlatform(boolean) – if true, lets trhe OS decide where to place the window
  + setVisible(boolean) – shows/hides the window
  + setDefaultCloseOperation() – what should the program do when the window closes
    - EXIT\_ON\_CLOSE – entire program exits when window closes

Adding components

* JFrame is a container – it can hold other components inside it
* To access the main container in a JFrame, use getContentPane()
* Then add a JButton
  + JButton constructor sets the button label

Components

* JLabel
* JTextField (can enter text)
* JList
* JCheckbox
* JRadioButton
* JComboBox
* Parent class – javax.swing.JComponent
* All have a setEnabled(boolean) method
  + When true: component is active and can be used
  + False: “greyed out”

JList in Modified MVC

* JList is the UI delegate (it is the view and the controller)
* It has an associated ListModel which provides the model (usually DefaultListModel used)
* ListModel is almost the same as java.util.List – methods include addElement(), remove(), insertElementAt(), getSize()
* When the ListModel is changed, info displayed is changed

Events in Swing

* An event is fired every time something happens in the program
  + JButton pressed, Window shown/hidden/opened/closed, user selects an item, …
* Event listeners
  + Will be called every time a relevant event happens
  + ActionListener, ComponentListener, MouseInputListener, ListSelectionListener, WindowListener

General strategy for Swing:

1. Instantiate the appropriate GUI components
2. Register callbacks for their interactive behaviour (register for all events that you might need to catch)
3. Add each component to the top-level window (e.g. JFrame)
4. Make the window visible

# Week 7

## Lecture 14: Threads (Interference, Liveness)

Concurrency:

* Process
  + Self-contained execution environment – own memory space
  + Often synonymous with programs/apps – however, an application may consist of several processes (e.g., Chrome)
* Thread
  + Lightweight processes: shared memory space
  + Every process has at least 1 thread
* Reasons to use
  + Tasks can be divided into subtasks; independent subtasks can be executed in parallel
  + Theoretical possible performance gain (Amdahl’s Law):
    - If F is the percentage of the program which cannot run in parallel and n is the number of processes, then the maximum performance gain is
* Reasons not to use:
  + Visibility
    - Thread A reads shared data which is later changed by thread B without A being aware of the change
  + Access
    - Several threads access and change shared data at the same time
  + Possible problems:
    - Liveness failure – program does not react any more due to problems in shared access
    - Safety failure – program creates incorrect data

Concurrency in Java:

* Java uses multithreaded programming within a single process
* Basic building block: Thread class
* Useful helper package: java.util.concurrent
* Threads
  + Creating
    - Preferred technique: implement the Runnable interface and define run() method
      * Can be subclassed, but not as flexible/general
    - Create thread based on Runnable class and use start()
  + Interrupting
    - From outside: call interrupt() on the Thrad object
    - Inside:
      * If calling a method that could throw InterruptedException (sleep(), join(), etc), return after it is caught
      * Otherwise, periodically check Thread.interrupted() and return if true
  + Pausing
    - Thread.sleep() with parameter for period of time
      * Lets other threads use system resources
      * Waits for something time-dependent to finish
    - Duration can be specified in milliseconds or nanoseconds – not guaranteed to be precise, depends on underlying OS and might be interrupted
    - Thread.sleep() throws InterruptedException (checked) – indicates that another thread interrupted the sleep and the thread should terminate
  + Waiting for a thread to terminate
    - Use join()
      * Can throw InterruptedException
      * Should be used at the end of any multithreaded program to be sure it terminates

Thread interference:

* Interference happens when 2 operations run in different threads but on the same interleave
  + 2 operations have multiple steps and the steps overlap
* Ex: i++ (retrieve i, increment, store in i)
* Avoiding: impose an ordering
  + Establish a happens-before relationship between 2 statements
    - Every statement before a Thread.start() happens before every statement executed by that thread
    - When a thread terminates and causes Thread.join() to return, every statement in the terminated thread happens before every statement following the join
  + Establish synchronized methods
    - Ensures that
      * Two calls to synchronized methods on the same object cannot interleave
      * When a synchronized method exits, it happens-before any other synchronized method on the same object
    - Constructors cannot be synchronized
  + Under the hood: Intrinsic locks
    - Every Java object has a lock associated with it
    - A thread that needs consistent access to an object field must acquire the lock before access and release the lock when it is done
      * In between, the thread owns the lock – no other thread can acquire it (will block on attempt)
      * A thread can access the same lock multiple times (re-entrant)
    - Synchronized methods make implicit use of the lock
    - More fine-grained option: synchronized statements

Atomic access:

* Atomic action – effectively happens all at once (cannot stop in the middle)
* Threads and writes are atomic for most types (except long and double)
* Increments (i++) are not atomic
* Avoids need for synchronized code

Liveness problems:

* Liveness: concurrent program’s ability to execute in a timely fashion
* Potential problems:
  + **Deadlock**: 2+ threads are blocked forever, waiting for each other
    - To fix (using simple locks): use consistent ordering on the acquisition of locks
  + Starvation: a thread cannot gain access to a shared resource and is unable to make progress
  + Livelock: threads too busy responding to each other to make progress

## Lecture 15: High Level Concurrency

Lock objects:

* Generalised version of synchronized code (simple intrinsic lock)
* Basic interface: java.util.concurrent.locks.Lock
  + Works like intrinsic locks
  + Only 1 thread can own a Lock object at a time
* Big advantage: allows code to back out of an attempt to acquire a lock
  + tryLock() – backs out if lock is not available or if timeout expires (timeout optional)
  + lockInterruptibly() – backs out if another thread sends interrupt before lock is acquired
* Best practice: put all code in a try-catch block and call lock.unlock() in a finally clause

Locks and Conditions:

* java.util.concurrent.locks.Condition – allows a thread to wait (not using any resources) until some condition is satisfied
* Get a Condition object from a Lock object with Lock.newCondition()
* Process
  + Acquire lock on object (lock(), tryLock(), lockInterruptibly(), etc)
  + If thread needs to wait, call condition.await()
  + When program is ready to continue, other thread calls condition.signal()
  + Do processing
  + Call lock.unlock()
* A single Lock can have multiple Condition objects to control different aspects

Atomic variables

* java.util.concurrent.atomic
* Defines classes that support atomic operations on single variables
  + All classes have get()/set() methods that impose happens-before – set happens before get
  + Atomic compareAndSet() method
  + Simple arithmetic methods that apply to integer atomic variables
    - decrementAndGet(), addAndGet(), getAndAdd(), …

Other useful Java libraries:

* java.util.concurrent: Concurrent collections
  + BlockingQueue: a FIFO structure that blocks when you attempt to add to a full queue or remove from an empty one
  + ConcurrentMap: defines atomic operations on maps (putIfAbsent, etc)
  + ConcurrentNavigableMap: supports approximate matches
* Streams support parallelStream() operator – processes Stream objects in parallel (Java runtime decides how to divide things up)
  + Any methods called in the context of a parallel stream must be thread-safe (locks, atomic, etc)

# Week 8

## Lecture 16: Streams

Overview:

* java.util.stream
* Primary Class: Stream<T> (generic)
  + Represents a sequence of elements
  + Allows operations to be performed on the sequence without needing to write, for example, loops or other control structures
  + All **intermediate** operations return a new Stream to allow operations to be **chained**
  + All **terminal** operations traverse the stream to produce a **result**/**side effect**
* Possible to get a Stream from a collection via stream(); other ways possible too (Stream.of, Arrays.stream, …)

Collection vs stream;

* Collection
  + Stores data internally
  + Operations modify data directly
  + Must be finite in size
* Stream
  + No storage – carry values from a source through a pipeline
  + Operations produce a new (modified) stream
  + Permits laziness
    - Elements can be returned on demand
    - Can represent infinite streams (e.g., all integers)

Lambda expressions

* Formally: they “express instances of single-method classes more succinctly”
* Structure:
  + Comma-separated list of formal parameters (can omit data type, can omits parentheses if only 1 parameter)
  + An arrow token “->”
  + A body consisting of a single expression or a statement block
* The argument to stream operations is a lambda expression
* Possible to use a method reference instead of a lambda expression
  + “ClassName::methodName”

Useful intermediate operations

* map(), mapToInt(), mapToDouble(), etc
  + Produces a new stream where the given operation has been applied to each element
* sorted()
  + Produces a new stream where the elements are sorted (assumes stream element implements Comparable)
* filter()
  + Produces a new stream containing only the elements that match
* limit()
  + Produces a new stream of at most the given number of elements
* distinct()
  + Produces a new stream of only unique objects according to equals()

Useful terminal operations:

* max(), min(), average(), sum()
  + Only on numeric streams
* count()
* collect(), toArray(), reduce()
  + Various ways of combining together all the elements of the stream
* findFirst(), findAny()
  + Tries to find an element matching the argument (returns an Optional)
* forEach
  + Implements internal iteration
  + Instead of using a “for” loop and implementing the body (**external iteration**), specify what should happen to each element and let the collection manage the details of processing the elements (**internal iteration**)

Useful methods to use in Collectors class:

* toList()
* toCollection()
* joining() – concatenate all elements as strings
* summingInt() – sum all integer values without going through an integer stream
* groupingBy() – collect into a Map (very powerful, read doc)
* partitioningBy() – collect into a Map, dividing into several lists (read doc)

Optional values

* Some terminal operations return an Optional<T> (e.g. findAny, findFirst)
* Represents a situation where the value might/might not exist, but returning null would be problematic
* How to deal with:
  + Check isPresent()
  + If true, access value with get()
  + Other option: use orElse to represent default value
* Ex: shapes.stream().filter(s -> s.getColor == YELLOW).findFirst();

## Lecture 17: Enumerated Types, Annotations, Immutable Classes

Enumerations

* A special data type that allows a variable to be one of a set of predefined constants
* Ex: Compass directions (NORTH, EAST, SOUTH, WEST), days in a week, months of a year, etc
* Declaration:
  + public enum Day { SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, }
  + Values are constants => written in all caps
  + An enum called Day should be in a class Day.java
  + Can be declared with arguments for constructor
* Special class
  + It has methods
    - Built-in static method **values()**: returns an array of all values
    - Built-in static method **valueOf()**: parses a string into an enum constant
    - Appropriate definitions of **compareTo(), equals(), hashCode(), toString()**
    - Other methods:
      * ordinal(): returns position of this constant in the list
      * name(): returns name of this constant
  + Fields can be defined, too
* How to use:
  + switch statements
  + create with valueOf
  + iterate with values()
  + accessing names, ordinal positions

Annotations:

* A form of metadata that provides data about a program that is not part of the program itself
* Have no direct effect on the operation of the code they annotate
* Uses:
  + Info for the compiler: detect errors, suppress warnings
  + Compile-time processing: use annotations to generate code/XML/etc
  + Runtime processing: some annotations are available
* Format
  + Start with “@” sign
  + Ex: @Override
  + Refer to the element following them
  + Must appear **outside** comments
  + May have arguments inside parentheses (if none, brackets can be omitted)
    - @SuppressWarnings(“unchecked”)
* Locations
  + Generally, applied to declarations (classes, fields, methods, etc)
  + Conventionally, each annotation on its own line
  + Annotations can be applied to the use of types
    - Ensures stronger type checking
    - Not built into Java itself, downloadable packages exist
    - Ex: @NonNull String str;
* Useful predefined ones:
  + @Deprecated: marks code as deprecated (still included but use is discouraged)
  + @Override: indicates that the labelled method must override a superclass method
    - Compiler produces an error message unless this is true
    - Automatically added by Eclipse whenever override/implement happens
  + @SuppressWarnings: disables particular compiler warnings
    - Argument indicates category:
      * deprecation: disable warning on use of deprecated method
      * unchecked: disable warning on use of non-generic code
    - Should be attached to innermost element where they apply
      * Do not disable warnings on a whole class if only 1 method is needed
  + All are defined in java.lang
* In Eclipse:
  + Automatically adds @Override over auto-generated methods where relevant
    - Interface implementation
    - Abstract class subclassing
    - Explicitly choosing “override/implement methods”
  + Proposes “quick fix” to suppress warnings

Immutable classes

* Immutable: internal state cannot change after it is constructed
* Ex: String, Wrapper classes (Integer, Long, Character, etc)
* Advantages of immutability:
  + Immutable objects can be safely shared between data structures or threads
  + Can save memory
  + Ideal for lookup keys in “dictionary” structures
* In practice:
  + Always a need to create new objects for new contents
    - No possibility to change state, e.g. setColor(RED)
  + More efficient to reuse objects
    - Slight performance difference
    - Decreased garbage collection overhead
    - No need for code to protect objects from corruption
* String operations
  + Lots of constructors and static initialisers
  + Lots of getters (charAt, indexOf, length)
  + Lots of state-checking methods (contains, compareTo, equalsIgnoreCase, startsWith)
  + Other methods return a new string (no modification) (concat, toLowerCase, replace, trim)
* Creation
  + Instance fields:
    - Must be private and final
    - Must have no setters, just getters
      * Don’t return mutable objects directly from getters, return copies
  + Constructor:
    - Must set complete internal state of object
    - Create copies of mutable parameters
  + Methods
    - Don’t allow overriding
      * declare class “final”
      * or make constructor “private” and use static factory methods to create instances