Java CheatSheet

Modern Java is a strongly-typed, eagery evaluated, case sensative, yet whitespace insensative language. It uses hierarchies of classes/types to structure data, but also has first-class support for functional-style algebraic datatypes.

Java programs are made up of 'classes', classes contain methods, and methods contain commands. To just try out a snippet of code, we can

- ◇ Open a terminal and enter `jshell`; then enter:

  ```
  1 + 2 // The jshell lets you try things out!

  // Say hello in a fancy way
  import javax.swing.*;
  JOptionPane.showMessageDialog(new JFrame(), "Hello, World!");
  ```

- ◇ Alternatively, in IntelliJ, click *Tools* then *Groovy Console* to try things out!
- ◇ Finally, VSCode allows arbitrary Java code to be sent to a `jshell` in the background(!) and it echoes the result in a friendly way.

To be terse, lots of content is not shown in this PDF, but is shown in the **HTML** version.

The Philosophy of Classes & Interfaces

Real life objects have properties and behaviour. For example, my cat has the properties of *name* and *age*, and amongst its behaviours are *sleep* and *meow*. However, an apple does not have such features. **The possible features of an object are understood when we *classify* it;** e.g., my cat is an animal, whereas an apple is food. In Java, a set of features is known as a `class` and objects having those features are called "objects of that class". Just as `int` is the type of the value `12`, we say a class is a *type* of an object.

We tend to think in (disjoint, hierarchical) categories; for example, in my library, a book can be found in one section, either "Sports" or "History" or "Trees". So where should books on the history of football be located? Or books on the history of trees? My library places such books under "Sports", and "Tree" respecively, but then adds a "historical" **tag** to them. Likewise, in Java, **to say different kinds of things have a feature in common, we "tag" them with an `interface`.** (Real life tagging is a also known as *multi*-`class`-*ificiation*.)

Java's Main Organisational Mechanisms

	With state	Without state
Attributes & properties	`class`	`record`
Partial implementations	`abstract class`	`interface`

Primitive Objects

For performance reasons, there are a handful of types whose values are created by *literals*; i.e., "What you see is what you get". (As such, primitives are a basic building block which cannot be broken apart; whereas non-primitives (aka references) are made-up from primitives and other references.) For example, to create a value of `int` we simply write `5`.

There are no instance methods on literals; only a handful of operator methods. For example, we cannot write `2.pow(3)` to compute 2^3, but instead must write `Math.pow(2, 3)`. Finally, variables of primitive types have default values when not initialised whereas object types default to `null` —note: `null` is a value of all object types, but not of primitive types.

```
// Declare a new object type
class Person { String name; }

Person obj; // ≈ null (OBJECT)
int prim;    // ≈ 0   (PRIMITIVE)

// Primitives are created as literals
prim = 1;    // ≈ 1

// Objects are created with "new"
obj = new Person(); // ≈ a reference,
   // like: Person@66048bfd

// Primitives are  identified by
// thier literal shape
assert prim == 1;

// Objects are identified by
/// references to their memory
// locations (not syntax shape!)
```

```
assert obj != new Person();

// Primitives copy values
int primCopy = prim;  // ≈ 1

/// Objects copy references
Person objCopy = obj;
  // ≈ a reference, like: Person@66048bfd

// Changing primitive copy has
// no impact on original
primCopy = 123;
assert prim == 1;

// Changing object copy also
// changes the original!
assert obj.name == null;
objCopy.name = "woah";    // Alter copy!
// Original is altered!
assert obj.name.equals("woah");
```

Properties and methods have separate namespaces

Properties and methods have **separate namespaces** —"Java is a Lisp-2 Language". Below we use the name `plus1` in two different definitional roles. Which one we want to refer to depends on whether we use "dot-notation" with *or* without parenthesis: The parentheis indicate we want to use the method.

```
class SameNameNoProblem {
    public static int plus1(int x){ return x + 1; } // Method!
    public static String plus1 = "+1";             // Property!
}

class ElseWhere {
    String pretty = SameNameNoProblem.plus1;
    Integer three = SameNameNoProblem.plus1(2);
}
```

The consequence of different namespaces are

1. Use `apply` to call functions bound to variables.
2. Refer to functions outside of function calls by using a double colon, `::`.

Let's discuss both of these now...

Anonymous *functions*: `(arg₁, ..., argₙ) → bodyHere`

Functions are formed with the "→" notation and used with "apply"

```
// define, then invoke later on
Function<Integer, Integer> f  =  x -> x * 2;

f.apply(3) // ⇒ 6
// f(3)    // invalid!

// define and immediately invoke
((Function<Integer, Integer>) x -> x * 2).apply(3);

// define from a method reference, using "::"
Function<Integer, Integer> f = SameNameNoProblem::plus1;
```

Let's make a method that takes anonymous functions, and use it

```
// Recursion with the 'tri'angle numbers: tri(f, n) = Σⁿᵢ₌₀ f(i).
public static int tri(Function<Integer, Integer> f, int n) {
    return n <= 0 ? 0 : f.apply(n) + tri(f, n - 1);
}

tri(x -> x / 2, 100);  //  ⇒  Σ¹⁰⁰ᵢ₌₀ i/2 = 2500

// Using the standard "do nothing" library function
tri(Function.identity(), 100);  //  ⇒  Σ¹⁰⁰ᵢ₌₀ i = 5050
```

Exercise! Why does the following code work?

```
int tri = 100;
tri(Function.identity(), tri); //  ⇒  5050

Function<Integer, Integer> tri = x -> x;
tri(tri, 100); //  ⇒  5050
```

In Java, everything is an object! (Ignoring primitives, which exist for the purposes of efficiency!) As such, functions are also objects! Which means, they must have a type: Either some class (or some interface), but which one? The arrow literal notation `x -> e` **is a short-hand** for an implementation of an interface with one abstract method…

Lambdas are a shorthand for classes that implement functional interfaces

Let's take a more theoretical look at anonymous functions.

Functional Interfaces

A *lambda expression* is a (shorthand) implementation of the only abstract method in a *functional interface* ——which is an interface that has exactly one abstract method, and possibly many default methods.

For example, the following interface is a functional interface: It has only one abstract method.

```
public interface Predicate<T> {

    boolean test(T t);  // This is the abstract method

    // Other non-abstract methods.
    default Predicate<T> and(Predicate<? super T> other) { ... }
    // Example usage: nonNull.and(nonEmpty).and(shorterThan5)
    static <T> Predicate<T> isEqual(T target) {...}
    // Example usage: Predicate.isEqual("Duke") is a new predicate to use.
}
```

Optionally, to ensure that this is indeed a functional interface, i.e., it has only one abstract method, we can place `@FunctionalInterface` above its declaration. Then the complier will check our intention for us.

The Type of a Lambda

Anyhow, since a lambda is a shorthand implementation of an interface, this means that what you can do with a lambda depenends on the interface it's impementing!

As such, when you see a lambda it's important to know it's type is not "just a function"! This mean **to run/apply/execute a lambda variable** you need to remember that the variable is technically an object implementing a specific functional interface, which has a single *named* abstract method (which is implemented by the lambda) and so we need to invoke that method on our lambda variable to actually run the lambda. For example,

```
Predicate<String> f = s -> s.length() == 3;   // Make a lambda variable
boolean isLength3String = f.test("hola");     // Actually invoke it.
```

Since different lambdas may implement different interfaces, the actually method to run the lambda will likely be different! Moreover, you can invoke *any* method on the interface that the lambda is implementing. After-all, a lambda is an object; not just a function.

Moreover, `Function` has useful methods: Such as `andThen` for composing functions sequentially, and `Function.identity` for the do-nothing function.

Common Java Functional Types

Anyhow, Java has ~40 functional interfaces, which are essentially useful variations around the following 4:

Class	runner	Description & example
`Supplier<T>`	`get`	Makes objects for us; e.g., `() -> "Hello"!` .
`Consumer<T>`	`accept`	Does stuff with our objects, returning void; e.g., `s -> System.out.println(s)` .
`Predicate<T>`	`test`	Tests our object for some property, returning a boolean e.g., `s -> s.length() == 3`
`Function<T`	`apply`	Takes our object and gives us a new one; e.g., `s -> s.length()`

For example, $\mathcal{C}$`::new` is a supplier for the class $\mathcal{C}$, and the forEach method on iterables actually uses a consumer lambda, and a supplier can be used to reuse streams (discussed below).

The remaining Java functional interfaces are variations on these 4 that are optimised for primitive types, or have different number of inputs as functions. For example, `UnaryOperator<T>` is essentially `Function<T, T>`, and `BiFunction<A, B, C>` is essentially `Function<A, Function<B, C>>` ———not equivalent, but essentially the same thing.

- As another example, Java has a `TriConsumer` which is the type of functions that have 3 inputs and no outputs —since `Tri` means 3, as in *tricycle*.

Eta Reduction: Writing Lambda Expressions as Method References

Lambdas can sometimes be simplified by using *method reference*:

Method type			
Static	$(x, ys) \to \tau.f(x, ys)$	$\approx$	$\tau :: f$
Instance	$(x, ys) \to x.f(ys)$	$\approx$	$\tau :: f$, where τ is the type of x
Constructor	`args` $\to$ `new` τ`<A>(args)`	$\approx$	τ`<A>::new`

For example, `(sentence, word) -> sentence.indexOf(word)` is the same as `String::indexOf`. Likewise, `(a, b) -> Integer.max(a, b)` is just `Integer::max`.

- Note that a class name τ might be qualified; e.g., `x -> System.out.println(x)` is just `System.out::println`.

Variable Bindings

Let's declare some new names, and assert what we know about them.

```
Integer x, y = 1, z;
assert x == null && y == 1 && z == null;
```

τ $\mathtt{x}_0$ `=` $\mathtt{v}_0$`, ..., ` $\mathtt{x}_n$ `=` $\mathtt{v}_n$`;` introduces n-new names $\mathtt{x}_i$ each having value $\mathtt{v}_i$ of type τ.

- The $\mathtt{v}_i$ are optional, defaulting to `0, false, '\000', null` for numbers, booleans, characters, and object types, respectively.
- Later we use $\mathtt{x}_i$ `=` $\mathtt{w}_i$`;` to update the name $\mathtt{x}_i$ to refer to a new value $\mathtt{w}_i$.

There are a variety of update statements: Suppose τ is the type of x then,

Augment:	`x ⊕= y`	$\approx$	`x = (τ)(x ⊕ y)`
Increment:	`x++`	$\approx$	`x += 1)`
Decrement:	`x--`	$\approx$	`x -= 1)`

The operators `--` and `++` can appear *before or after* a name: Suppose $\mathcal{S}(x)$ is a statement mentioning the name x, then

$\mathcal{S}$`(x++)`	$\approx$	$\mathcal{S}$`(x); x += 1`
$\mathcal{S}$`(++x)`	$\approx$	`x += 1;` $\mathcal{S}$`(x)`

Since compound assignment is really an update with a *cast*, there could be unexpected behaviour when x and y are not both ints/floats.

- If we place the keyword `final` before the type τ, then the names are constant: They can appear only once on the right side of an '=', and any further occurrences (i.e., to change their values) crash the program. `final int x = 1, y; y = 3;` is fine, but changing the second `y` to an `x` fails.
- We may use `var x = v`, for only *one* declaration, to avoid writing the name of the type τ (which may be lengthy). Java then *infers* the type by inspecting the shape of `v`.
- Chained assignments associate to the right:

  `a += b /= 2 * ++c;` $\approx$ `a += (b /= (2 * ++c));`

  (The left side of an "=", or "⊕=", must a single name!)

Strings

Any pair of matching double-quotes will produce a string literal —whereas single-quote around a single character produce a `char`acter value. For multi-line strings, use triple quotes, `"""`, to produce *text blocks.*

String interpolation can be done with `String.format` using `%s` placeholders. For advanced interpolation, such as positional placeholders, use MessageFormat.

```
String.format("Half of 100 is %s", 100 / 2) // ⇒ "Half of 100 is 50"
```

- `s.repeat(`n`)` $\approx$ Get a new string by gluing n-copies of the string $\int$.
- `s.toUpperCase()` and `s.toLowerCase()` to change case.
- Trim removes spaces, newlines, tabs, and other whitespace from the start and end of a string. E.g., `"  okay \n ".trim().equals("okay")`
- `s.length()` is the number of characters in the string.
- `s.isEmpty()` $\equiv$ `s.length() == 0`
- `s.isBlank()` $\equiv$ `s.trim().isEmpty()`
- `String.valueOf(x)` gets a string representation of anything `x`.
- `s.concat(t)` glues together two strings into one longer string; i.e., `s + t`.

Equality

- In general, '==' is used to check two primitives for equality, whereas `.equals` is used to check if two objects are equal.
- The equality operator '==' means "two things are indistinguishable: They evaluate to the same literal value, or refer to the same place in memory".
- As a method, `.equals` can be redefined to obtain a suitable notion of equality between objects; e.g., "two people are the same if they have the same name (regardless of anything else)". If it's not redefined, `.equals` behaves the same as '=='. In contrast, Java does not support operator overloading and so '==' cannot be redefined.
- For strings, '==' and `.equals` behave differently: `new String("x") == new String("x")` is false, but `new String("x").equals(new String("x"))` is true! The first checks that two things refer to the same place in memory, the second checks that they have the same letters in the same order.
  - If we want this kind of "two objects are equal when they have the same contents" behaviour, we can get it for free by using `record`s instead of `class`es.

Arithmetic

In addition to the standard arithmetic operations, we have `Math.max(x, y)` that takes two numbers and gives the largest; likewise `Math.min(x, y)`. Other common functions include `Math.sqrt`, `Math.ceil`, `Math.round`, `Math.abs`, and `Math.random()` which returns a random number between 0 and 1. Also, use `%` for remainder after division; e.g., `n % 10` is the right-most digit of integer n, and `n % 2 == 0` exactly when n is even, and `d % 1` gives the decimal points of a floating point number d, and finally: If `d` is the index of the current weekday (0..6), then `d + 13 % 7` is the weekday 13-days from today.

```
// Scientific notation: xey ≈ x × 10^y
assert 1.2e3 == 1.2 * Math.pow(10, 3)
```

```
// random integer x with 4 ≤ x < 99
var x = new Random().nextInt(4, 99);
```

Sum the digits of the integer $n = 31485$

```
int n = 31485;
int sum = 0;
while (n % 10 != 0) { sum += n % 10; n /= 10; }
assert sum == 3 + 1 + 4 + 8 + 5;
```

A more elegant, "functional style", solution:

```
String.valueOf(n).chars().map(c -> c - '0').sum();
```

The `chars()` methods returns a stream of integers (Java characters are really just integers). Likewise, `IntStream.range(0, 20)` makes a sequence of numbers that we can then `map` over, then `sum`, `min`, `max`, `average`.

Collections and Streams

Collections are types that hold a bunch of similar data: Lists, Sets, and Maps are the most popular. *Streams* are pipelines for altering collections: Usually one has a collection, converts it to a stream by invoking `.stream()`, then performs `map` and `filter` methods, etc, then "collects" (i.e., runs the stream pipeline to get an actual collection value back) the result.

Lists are ordered collections, that care about multiplicity. Lists are made with `List.of(`x_0`,` x_1`, ...,` x_n`)`. Indexing, `xs.get(`i`)`, yields the i-th element from the start; i.e., the number of items to skip; whence `xs.get(0)` is the first element.

Sets are unordered collections, that ignore multiplicity. Sets are made with `Set.of(`x_0`,` x_1`, ...,` x_n`)`.

Maps are pairs of 'keys' along with 'values'. `Map<K, V>` is essentially the class of objects that have no methods but instead have an arbitary number of properties (the 'keys' of type `K`), where each property has a value of type `V`. Maps are made with `Map.of(`k_0`,` v_0`, ...,` k_{10}`,` v_{10}`)` by explicitly declaraing keys and their associated values. The method $\mathcal{M}$`.get(k)` returns the value to which the specified key `k` is mapped, or `null` if the map $\mathcal{M}$ contains no mapping for the key. Maps have an `entrySet()` method that gives a set of key-value pairs, which can then be converted to a stream, if need be.

Other collection methods include, for a collection instance $\mathcal{C}$:

- $\mathcal{C}$`.size()` is the number of elements in the collection
- $\mathcal{C}$`.isEmpty()` $\equiv$ $\mathcal{C}$`.size() == 0`
- $\mathcal{C}$`.contains(e)` $\equiv$ $\mathcal{C}$`.stream().filter(x -> x.equals(e)).count() > 0`
- `Collections.fill(`$\mathcal{L}$`, e)` $\cong$ $\mathcal{L}$`.stream().map(_ -> e).toList()`; i.e., copy list $\mathcal{L}$ but replace all elements with `e`.
- `Collections.frequency(`$\mathcal{C}$`, e)` counts how many times `e` occurs in a collection.
- `Collections.max(`$\mathcal{C}$`)` is the largest value in a collection; likewise `min`.
- `Collections.nCopies(n, e)` is a list of n copies of `e`.

`Stream<`τ`>` methods

- `Stream.of(`x_0`, ...,` x_n`)` makes a stream of data, of type τ, ready to be acted on.
- `s.map(f)` changes the elements according to a function $f : \tau \to \tau'$.
  - `s.flatMap(f)` transforms each element into a stream since $f : \tau \to Stream < \tau' >$, then the resulting stream-of-streams is flattened into a single sequential stream.
  - As such, to merge a streams of streams just invoke `.flatMap(s -> s)`.
- `s.filter(p)` keeps only the elements that satisfy property `p`
- `s.count()` is the number of elements in the stream
- `s.allMatch(p)` tests if all elements satisfy the predicate `p`
- `s.anyMatch(p)` tests if any element satisfies `p`
- `s.noneMatch(p)` $\equiv$ `s.allMatch(p.negate())`
- `s.distinct()` drops all duplicates
- `s.findFirst()` returns an `Optional<`τ`>` denoting the first element, if any.
- `s.forEach(a)` to loop over the elements and perform action `a`.
  - If you want to do some action, and get the stream `s` back for further use, then use `s.peek(a)`.

Generics

Java only lets us return a single value from a method, what if we want to return a pair of values? Easy, let's declare `record Pair(Object first, Object second) { }` and then return `Pair`. This solution has the same problem as methods that just return `Object`: It communicates essentially no information —after all, *everything is an object!*— and so requires dangerous casts to be useful, and the compiler wont help me avoid type mistakes.

```
record Pair(Object first, Object second) { }

// This should return an integer and a string
Pair myMethod() { return new Pair("1", "hello"); } // Oops, I made a typo!

int num = (int) (myMethod().first()); // BOOM!
```

It would be better if we could say "this method returns a pair of an integer and a string", for example. We can do just that with *generics*!

```
record Pair<A, B>(A first, B second) { }

Pair<Integer, String> myMethod() { return new Pair<>(1, "hello"); }

int num = myMethod().first();
```

This approach *communicates to the compiler my intentions* and so the compiler ensures I don't make any silly typos. Such good communication also means no dangerous casts are required.

We can use the new type in three ways:

`Pair<A, B>`	explicitly providing the types we want to use `Pair` with
`Pair<>`	letting Java *infer, guess,* the types for `Pair` by how we use it
`Pair`	defaulting the types to all be `Object`

The final option is not recommended, since it looses type information. It's only allowed since older versions of Java do not have type parameters and so, at run time, all type parameters are 'erased'. That is, *type parameters only exist at compile time and so cannot be inspected/observed at run-time.*