FROM JAVA 8 PERSPECTIVE

Lambda expression ⇒ way to represent anonymous fn's.

e.g 1 Associate behaviour with a button click

```
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        S.o.p("button clicked");
    }
});
```

The construct is obscure as we want to pass behaviour but we are passing objects instead.

Soln
```
button.addActionListener(event -> S.o.p("button clicked"));
```
Instead of passing an object of an interface, a fn without a name is passed. → separates the parameter from the body of the lambda expression. The type ActionEvent is not mentioned in the syntax but javac infers the type of variable event from its context (signature of addActionListener)

Some more variations :-

e.g 2 `Runnable noArguments = () -> S.o.p("Hello World");`
Lambda expression to implement Runnable interface where the only method run() takes no arguments.

e.g 3. `ActionListener oneArgument = event -> S.o.p("button clicked");`
One argument lambda — no parantheses.

e.g 4. `Runnable multiStatement = () -> { S.o.p("Hello"); S.o.p("World"); };`
return or throw exceptions are also possible.

e.g 5. `BinaryOperator<Long> addExplicit = (Long x, Long y) -> x+y;`
In all these examples type is inferred from the context just like

```
final String[] array = {"hello", "world"};
```
(type is inferred)

Using null is another example of type inference.

Immutable Values :-

Anonymous inner classes can only access final variables of their surrounding methods.

e.g 6

```
final String name = getUserName();
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent event){
        S.o.p("Hello" + name);
    }
});
```

```
String name = getUserName();
button.addActionListener(event -> S.o.p("Hello" + name));
```

Free variable captured by lambda

→ name is effectively final even if not explicitly final.

This explains closures ⇒ the variables that are not assigned to are closed over the surrounding state in order to bind them to a value. (Lambdas close over values rather than variables.)

Functional Interface :-

An interface with a single abstract method that is used as the type of lambda expression.

Functional interfaces (i) may use more than one parameter and return a value (ii) may use generics. (iii) signature is same as the abstract method of the functional interface

ActionEvent → ActionListener → ~~return type~~

ActionListener interface, Runnable, Comparator interfaces are functional interfaces.

~~event~~ → (x, y) → { ~~return~~ x+y ; }

Lambda expressions are still type checked and safe as javac infers the type of lambda expressions from the information provided. This is what is meant by type safety.

e.g. 7

```
Predicate<Integer> atLeast5 = x -> x > 5;
```

Predicate<String> nEStr = (String s) -> !s.isEmpty();

```
public interface Predicate<T>{
    boolean test(T t);
}
```

T → Predicate → boolean

introduced in Java 8

```
public interface Adder {
    int add(int x, int y);
}
```
↓ Functional interface

```
public interface SmartAdder extends Adder {
    int add(double a, double b);
}
```
↓ Two abstract methods
Not functional interface

Date 16/01/2017

e.g 8 `BinaryOperator<Long> addLongs = (x,y) → x+y;` ✓

`BinaryOperator add = (x,y) → x+y;` X Compile error.

Lack of sufficient information to predict the correct type.

error says → it cannot be applied to Objects

As no specific type is mentioned, compiler treats the parameters as Object and + does not work on Objects.

(*) Lamda expressions is a ~~method~~ function without a name that is used to pass around behaviours as if it were data.

It allows fns to be treated as data values like

(int x) → x+1 ⇒ the fn when called with argument x, returns x+1.

Features :- (i) Do not have a specific name.

(ii) Are not associated with any class like a java method.

(iii) Can be passed as an argument to a method & stored as a variable (passed around).

(iv) No need to be verbose like inner classes. (Concise)

Some more examples

(i) `(int x, int y) → { S.o.p("Result is "); S.o.p(x+y); }`

(ii) `() → { return "Mario"; }`

(iii) `(Integer i) → return "Alan" + i;` X ④

`(Integer i) → { return "Alan" + i; }` Since return is a control flow statement.

(iv) `(String s) → { "Iron Man"; }` X Not a valid statement

`(String s) → { return "Iron Man"; }`

Lamdas can be used to create ① objects, writing ② boolean expressions, extracting ③ data from an object, combine ④ two values and compare ⑤ two objects.

① `() → new Apple(10);`

② `(List<String> list) → list.isEmpty();`

③ `(String s) → s.length();`

⑤ `(Apple a1, Apple a2) → a1.getWeight().compareTo(a2.getWeight());`

Functional Interface

```
Runnable r1 = () → S.o.p("Thread 1");
Runnable r2 = new Runnable() {
                public void run() { S.o.p("Thread 2"); } };
public static void process(Runnable r) {
    r.run(); }
process(r1);
process(r2);
process(() → S.o.p("Thread 3"));
```

Lamda expr. as method parameter

Lamda expressions let programmers provide the implementation of the abstract method of the functional interface directly inline, and then treat the whole expression as an instance of the functional interface.

Java 8 defines several new functional interfaces inside java.util.function. e.g Predicate, Consumer, Function.

Java supports both primitive and reference types. Generic types are only defined for reference types. Boxing converts primitive type ⇒ ref type

Unboxing " ref type ⇒ primitive type

Autoboxing automatically performs boxing and/unboxing

```
List<Integer> list = new ArrayList<>();
for (int i = 300; i < 400; i++)
    list.add(i);
```

Each element of a primitive array is the size of the primitive.
Each element of a boxed array is an in memory pointer to another object on the java heap.

Primitive types are wrapped around (aka autoboxing) and stored in a heap. Boxed values require more memory and require additional lookup to get the wrapped primitive value.

```
public interface IntPredicate {
    boolean test(int t);
}
```

```
IntPredicate evenN = (int i) → i%2==0;
evenN.test(1000); — No boxing
```

Library differentiates bet² primitive and boxed versions of some library fn's. Only the int, long and double types have been chosen as the focus because the impact is most noticeable in numerical algorithms.

T → [ToLongFunction] → long long → [LongFunction] → T long → [LongBinaryOperator] → long

Predicate<Integer> oddN = (Integer i) → i%2 != 0;
oddN(1000); — boxing

More examples in Java 8 : DoublePredicate, IntConsumer, LongBinaryOper

The functional interfaces do not allow checked excep²s to be thrown. Those code snippets need to be kept within try-catch blocks.

User defined interfaces may declare to throw checked excep²s.

```
public interface BufferedReaderProcessor {
    public String process(BufferedReader b) throws IOExcep²;}
BufferedReaderProcessor p = (BufferedReader b) → b.readLine();
```

Lambda expressions same but mapped differently due to Type inferance

```
Callable<Integer> c = () → 42;
PriviledgedAction<Integer> p = () → 42;
```

declares fn that accepts nothing but returns generic T

As the fn descriptor is available through target type, appropriate signature for Lambda can be deduced.

Method Reference –

Syntactic sugar for lambda expressions to forward arguments.

```
Function<String, Integer> f = (String s) → Integer.parseInt(s);
Function<String, Integer> f = Integer::parseInt;
```

Overload resolution

```
private void overloaded(Object o) {
    s.o.p("Object");}
overloaded("abc");
```

```
private void overloaded(String o) {
    s.o.p("String");
```

javac will infer most specific type.

```
private interface IntPredicate {
    public boolean test(int value);}
private overloaded(Predicate<Integer> p) {
    s.o.p("Predicate");}
private overloaded(IntPredicate p) {
    s.o.p("IntPredicate");}
overloaded((x) → true);
```

javac will fail to compile as (x)→true may map to both versi of overloaded methods → ambiguou there is no most specific target type

Rules :–

(1) if there is a single specific target type, javac infers the type from the corresponding argument in the functional interface.

(2) if there are several specific target types, the most specific type is inferred.

(3) " " " . " . " . " and no " " " , manually provide the type.

(Now discuss about checked exceptions)

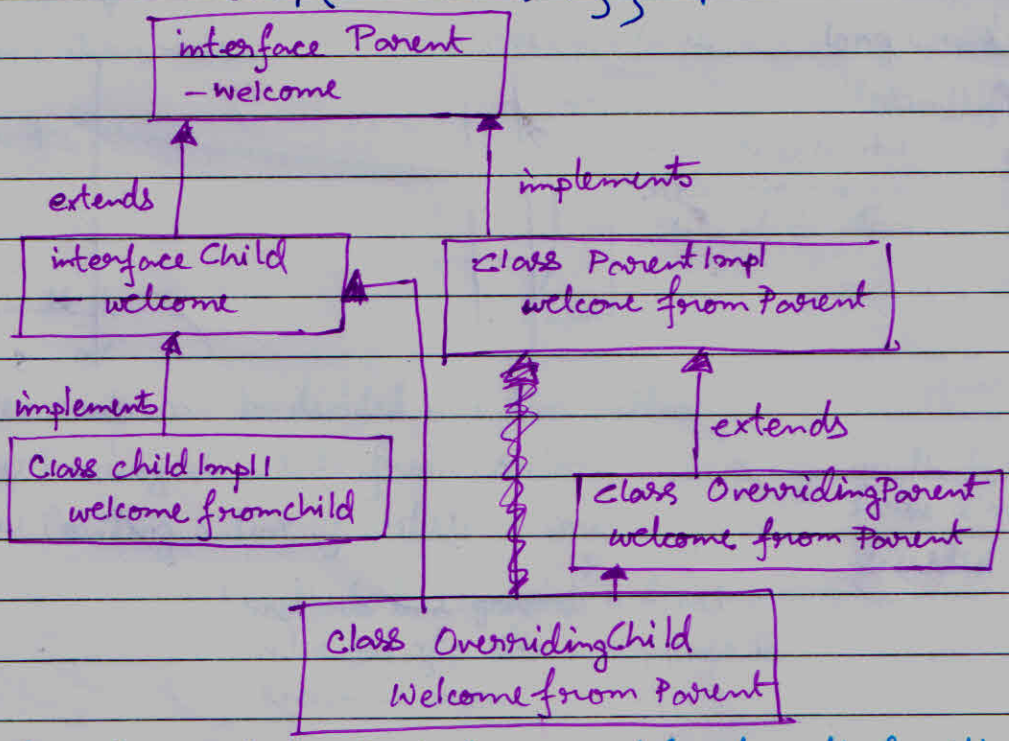
Not all interfaces with a single method are functional interfaces. e.g java.io.Closeable. If an object is closeable, it must hold a file object, a handle that can be closed at some point of time ⇒ mutating state (closing a resource)

~~Functional~~ javac checks for such fn syntax if @FunctionalInterface annotation is used.

stream() method is introduced in java 8 collection interface. But for backward compatibility with java 7 codes (say) default methods concept is introduced. default methods can be defined in interfaces.

```
public interface Parent {
    public default void welcome() {
        S.o.p("Welcome");}}
```

Interfaces in Java 8 dont have constant fields.



Rules

(1) if there's a method with a body or an abstract method with a superclass chain, ignore interface version of the default method.

(2) if two interfaces one extending the other compete for a default method, sub interface wins.

(3) in case of ambiguity, subclass must either implement the method or

declare it abstract. ⇒

```
interface 1 {
    default method1(){
    }
}
```

```
interface 2 {
    default method1(){
    }
}
```

~~call in~~ class ... implements interface1, interface2{

:

}

— ~~The~~ Ambiguity should be resolved following rule 3.

Thus multiple inheritance of state can be avoided in Java 8.