6. Java Errors and Exceptions

Errors, runtime-exceptions, checked-exceptions, exception handling, special case: resources

Errors and Exceptions in Java

Errors and exceptions interrupt the normal execution of the program abruptly and represent an *unplanned event*.

Exceptions are bad, or not?

- Java allows to catch such events and deal with it (as opposed to crashing the entire program)
- Unhandled errors and exceptions are passed up through the call stack.

Errors

This glass is broken for good

Errors happen in the virtual machine of Java and are *not repairable*.

Examples

- No more memory available
- Too high call stack ($\rightarrow$ recursion)
- Missing libraries
- Bug in the virtual machine
- Hardware error

Exceptions

Exceptions are triggered by the virtual machine or the program itself and can typically be handled in order to *re-establish the normal situation*

Clean-up and pour in a new glass

Examples

- De-reference `null`
- Division by zero
- Read/write errors (on files)
- Errors in business logic

Exception Types

Runtime Exceptions

- Can happen anywhere
- *Can* be handled
- Cause: bug in the code

Checked Exceptions

- Must be declared
- *Must* be handled
- Cause: Unlikely but not impossible event

Exception Types

Runtime Exceptions

- Can happen anywhere
- *Can* be handled
- Cause: bug in the code

Checked Exceptions

- Must be declared
- *Must* be handled
- Cause: Unlikely but not impossible event

Example of a Runtime Exception

```java
import java.util.Scanner;
class ReadTest {
    public static void main(String[] args){
        int i = readInt("Number");
    }
    private static int readInt(String prompt){
        System.out.print(prompt + ": ");
        Scanner input = new Scanner(System.in);
        return input.nextInt();
    }
}
```

Input: `Number: asdf`

Unhandled Errors and Exceptions

The program crashes and leaves behind a *stack trace*. In there, we can see the where the program got interrupted.

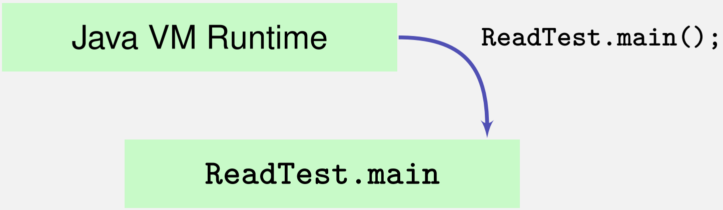
```
Exception in thread "main" java.util.InputMismatchException
            [...]
        at java.util.Scanner.nextInt(Scanner.java:2076)
        at ReadTest.readInt(ReadTest.java:9)
        at ReadTest.main(ReadTest.java:4)
```

⇒ Forensic investigation based on this information.

Exception gets Propagated through Call Stack

Java VM Runtime

Exception gets Propagated through Call Stack

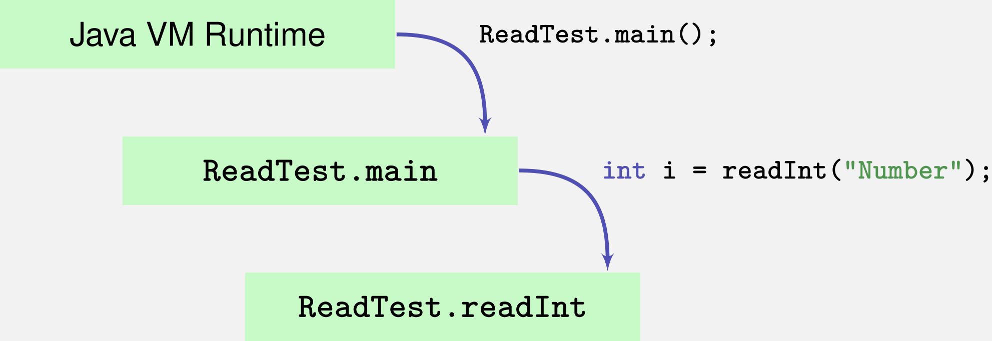


Java VM Runtime

`ReadTest.main();`

`ReadTest.main`

Exception gets Propagated through Call Stack



Java VM Runtime

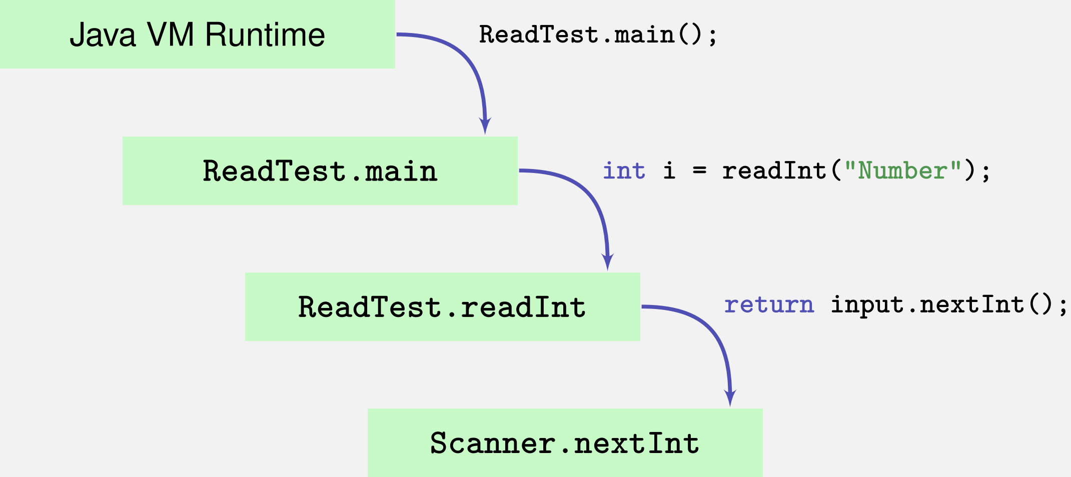
```
ReadTest.main();
```

ReadTest.main

```
int i = readInt("Number");
```

ReadTest.readInt

Exception gets Propagated through Call Stack



Java VM Runtime

```
ReadTest.main();
```

`ReadTest.main`

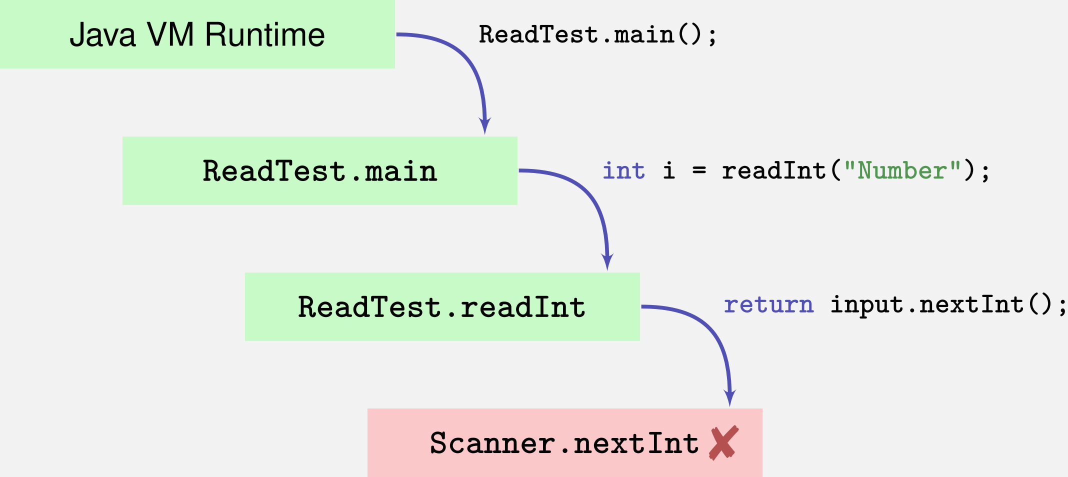
```
int i = readInt("Number");
```

`ReadTest.readInt`

```
return input.nextInt();
```

`Scanner.nextInt`

Exception gets Propagated through Call Stack



Java VM Runtime

`ReadTest.main();`

`ReadTest.main`

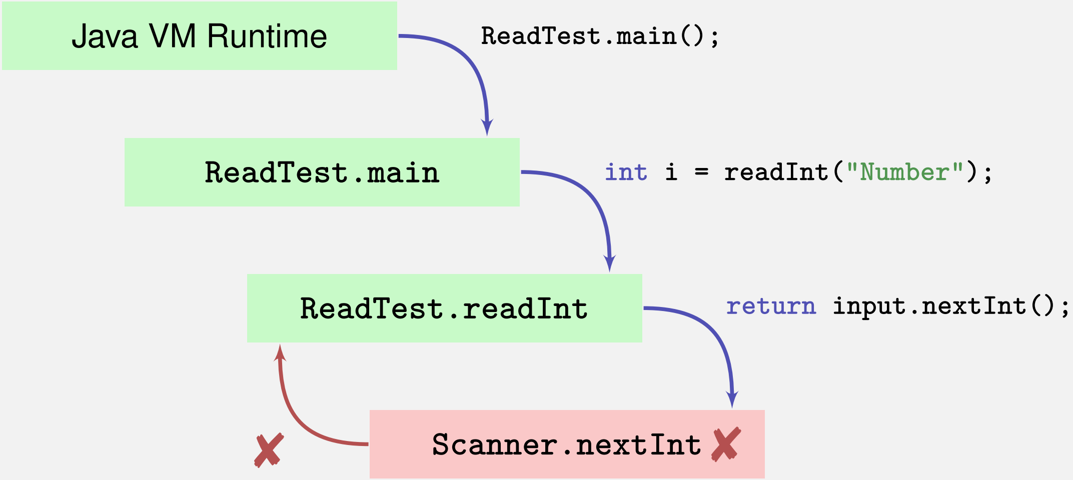
`int i = readInt("Number");`

`ReadTest.readInt`

`return input.nextInt();`

`Scanner.nextInt` ✗

Exception gets Propagated through Call Stack



Java VM Runtime

```
ReadTest.main();
```

`ReadTest.main`

```
int i = readInt("Number");
```

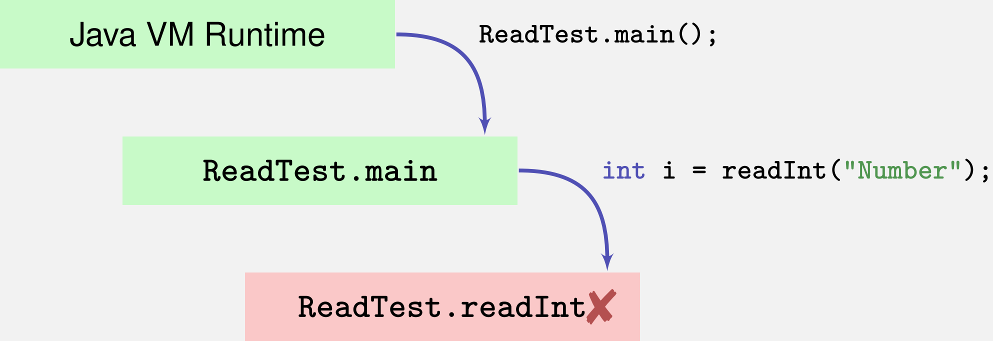
`ReadTest.readInt`

```
return input.nextInt();
```

`Scanner.nextInt` ✗

✗

Exception gets Propagated through Call Stack



Java VM Runtime

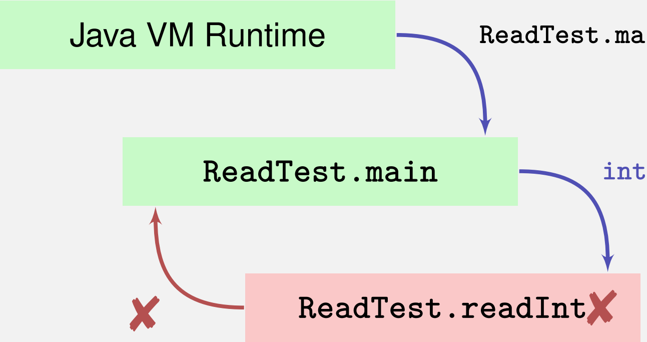
`ReadTest.main();`

`ReadTest.main`

`int i = readInt("Number");`

`ReadTest.readInt`✗

Exception gets Propagated through Call Stack



Java VM Runtime

`ReadTest.main();`

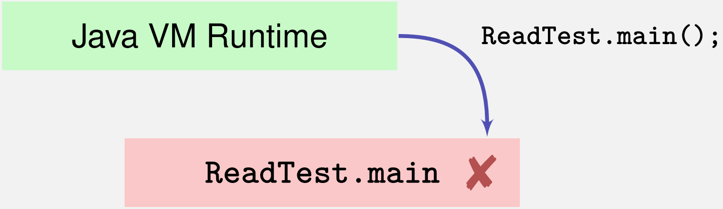
ReadTest.main

```
int i = readInt("Number");
```

ReadTest.readInt ✗

✗

Exception gets Propagated through Call Stack

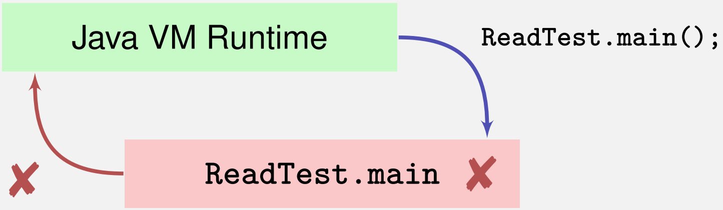


Java VM Runtime

`ReadTest.main();`

`ReadTest.main` ✗

Exception gets Propagated through Call Stack



Java VM Runtime

`ReadTest.main();`

`ReadTest.main` ✗

✗

Exception gets Propagated through Call Stack

Java VM Runtime †

Unstanding Stack Traces

Output:

```
Exception in thread "main" java.util.InputMismatchException
        at java.util.Scanner.throwFor(Scanner.java:864)
        at java.util.Scanner.next(Scanner.java:1485)
        at java.util.Scanner.nextInt(Scanner.java:2117)
        at java.util.Scanner.nextInt(Scanner.java:2076)
        at ReadTest.readInt(ReadTest.java:9)
        at ReadTest.main(ReadTest.java:4)
```

Unstanding Stack Traces

An unsuited input ...

Output:

```
Exception in thread "main" java.util.InputMismatchException
        at java.util.Scanner.throwFor(Scanner.java:864)
        at java.util.Scanner.next(Scanner.java:1485)
        at java.util.Scanner.nextInt(Scanner.java:2117)
        at java.util.Scanner.nextInt(Scanner.java:2076)
        at ReadTest.readInt(ReadTest.java:9)
        at ReadTest.main(ReadTest.java:4)
```

Unstanding Stack Traces

Output:

```
Exception in thread "main" java.util.InputMismatchException
        ~~at java . util .Scanner.throwFor(Scanner.java:864)~~
        ~~at java . util .Scanner.next(Scanner.java:1485)~~
        ~~at java . util .Scanner.nextInt(Scanner.java:2117)~~
        ~~at java . util .Scanner.nextInt(Scanner.java:2076)~~
        at ReadTest.readInt(ReadTest.java:9)
        at ReadTest.main(ReadTest.java:4)
```

Unstanding Stack Traces

Output:

```
Exception in thread "main" java.util.InputMismatchException
        at java.util.Scanner.throwFor(Scanner.java:864)
        at java.util.Scanner.next(Scanner.java:1485)
        at java.util.Scanner.nextInt(Scanner.java:2117)
        at java.util.Scanner.nextInt(Scanner.java:2076)
        at ReadTest.readInt(ReadTest.java:9)
        at ReadTest.main(ReadTest.java:4)
```

... in method `readInt` *on line 9 ...*

Unstanding Stack Traces

Output:

```
Exception in thread "main" java.util.InputMismatchException
        at java.util.Scanner.throwFor(Scanner.java:864)
        at java.util.Scanner.next(Scanner.java:1485)
        at java.util.Scanner.nextInt(Scanner.java:2117)
        at java.util.Scanner.nextInt(Scanner.java:2076)
        at ReadTest.readInt(ReadTest.java:9)
        at ReadTest.main(ReadTest.java:4)
```

... called by method `main` *on line 4.*

Unstanding Stack Traces

```
 1 import java.util.Scanner;
 2 class ReadTest {
 3     public static void main(String[] args){
 4         int i = readInt("Number");
 5     }
 6     private static int readInt(String prompt){
 7         System.out.print(prompt + ": ");
 8         Scanner input = new Scanner(System.in);
 9         return input.nextInt();
10     }
11 }
```

```
at ReadTest.readInt(ReadTest.java:9)
at ReadTest.main(ReadTest.java:4)
```

Unstanding Stack Traces

```
1  import java.util.Scanner;
2  class ReadTest {
3      public static void main(String[] args){
4          int i = readInt("Number");
5      }
6      private static int readInt(String prompt){
7          System.out.print(prompt + ": ");
8          Scanner input = new Scanner(System.in);
9          return input.nextInt();
10     }
11 }
```

```
at ReadTest.readInt(ReadTest.java:9)
at ReadTest.main(ReadTest.java:4)
```

Runtime Exception: Bug in the Code?!

Where is the bug?

```
private static int readInt(String prompt){
    System.out.print(prompt + ": ");
    Scanner input = new Scanner(System.in);
    return input.nextInt();
}
```

Runtime Exception: Bug in the Code?!

Where is the bug?

```
private static int readInt(String prompt){
    System.out.print(prompt + ": ");
    Scanner input = new Scanner(System.in);
    return input.nextInt();
}
```

Not guaranteed that the next input is an `int`

Runtime Exception: Bug in the Code?!

Where is the bug?

```
private static int readInt(String prompt){
    System.out.print(prompt + ": ");
    Scanner input = new Scanner(System.in);
    return input.nextInt();
}
```

Not guaranteed that the next input is an `int`

⇒ The scanner class provides a test for this

Runtime Exception: Bug Fix!

Check first!

```
private static int readInt(String prompt){
    System.out.print(prompt + ": ");
    Scanner input = new Scanner(System.in);
    if (input.hasNextInt()){
        return input.nextInt();
    } else {
        return 0; // or do something else ...?!
    }
}
```

First Finding: often no Exceptional Situation

Often, those “exceptional” cases aren’t that unusual, but pretty foreseeable. In those cases *no* exceptions should be used!

Kids are tipping over cups. You get used to it.

Examples

- Wrong credentials when logging in
- Empty required fields in forms
- Unavailable internet resources
- Timeouts

Second Finding: Avoid Exceptions

Problem solved.

Instead of letting a runtime exception happen, *actively prevent* such a situation to arise.

Examples

- Check user inputs early
- Use optional types
- Predict timeout situations
- Plan B for unavailable resources

Exception Types

Runtime Exceptions

- Can happen anywhere
- *Can* be handled
- Cause: bug in the code

Checked Exceptions

- Must be declared
- *Must* be handled
- Cause: Unlikely but not impossible event

Exception Types

Runtime Exceptions

- Can happen anywhere
- *Can* be handled
- Cause: bug in the code

Checked Exceptions

- Must be declared
- *Must* be handled
- Cause: Unlikely but not impossible event

Example of a Checked Exception

```
private static String[] readFile(String filename){
    FileReader fr = new FileReader(filename);
    BufferedReader bufr = new BufferedReader(fr);
    ...
    line = bufr.readLine();
    ...
}
```

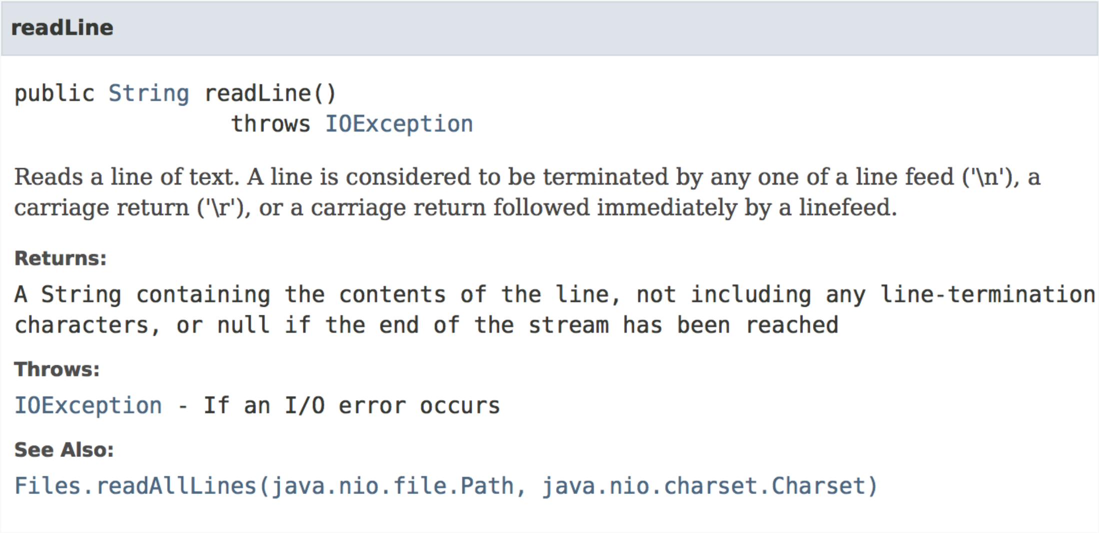
Example of a Checked Exception

```java
private static String[] readFile(String filename){
    FileReader fr = new FileReader(filename);
    BufferedReader bufr = new BufferedReader(fr);
    ...
    line = bufr.readLine();
    ...
}
```

Compiler Error:

```
./Root/Main.java:9: error: unreported exception FileNotFoundException; must be caught or declared to be
    FileReader fr = new FileReader(filename);
                    ^
./Root/Main.java:11: error: unreported exception IOException; must be caught or declared to be thrown
    String line = bufr.readLine();
                               ^
2 errors
```

Quick Look into Javadoc



readLine

```
public String readLine()
                throws IOException
```

Reads a line of text. A line is considered to be terminated by any one of a line feed ('\n'), a carriage return ('\r'), or a carriage return followed immediately by a linefeed.

Returns:

A String containing the contents of the line, not including any line-termination characters, or null if the end of the stream has been reached

Throws:

IOException - If an I/O error occurs

See Also:

Files.readAllLines(java.nio.file.Path, java.nio.charset.Charset)

Why use Checked Exceptions?

The following situations justify checked exception:

- Fault is *unprobable but not impossibe* – and can be fixed by taking suitable measures at runtime.

Why use Checked Exceptions?

The following situations justify checked exception:

- Fault is *unprobable but not impossibe* – and can be fixed by taking suitable measures at runtime.

The caller of a method with a declared checked exception is forced to deal with it – catch it or pass it up.

Handling Exceptions

```
private static String[] readFile(String filename){
    try{
        FileReader fr = new FileReader(filename);
        BufferedReader bufr = new BufferedReader(fr);
        ...
        line = bufr.readLine();
        ...
    } catch (IOException e){
        // do some recovery handling
    } finally {
        // close resources
    }
}
```

← *Protected scope*

Handling Exceptions

```java
private static String[] readFile(String filename){
    try{
        FileReader fr = new FileReader(filename);
        BufferedReader bufr = new BufferedReader(fr);
        ...
        line = bufr.readLine();
        ...
    } catch (IOException e){
        // do some recovery handling
    } finally {
        // close resources
    }
}
```

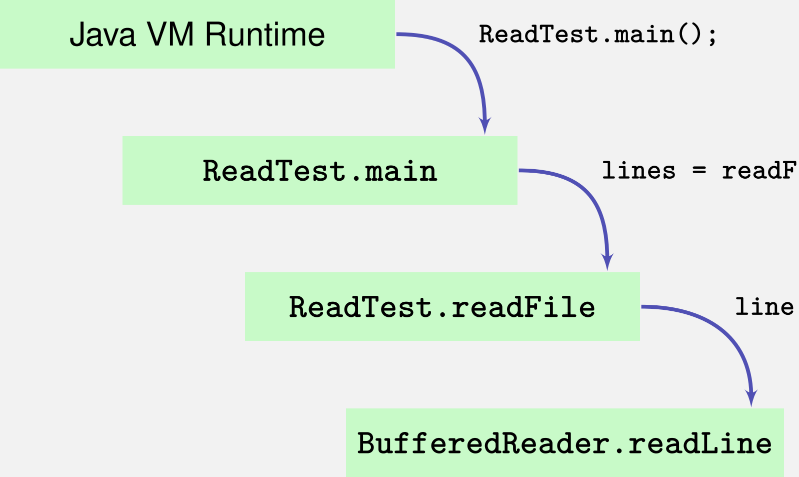
Measures to re-establis the normal situation

Handling Exceptions

```
private static String[] readFile(String filename){
    try{
        FileReader fr = new FileReader(filename);
        BufferedReader bufr = new BufferedReader(fr);
        ...
        line = bufr.readLine();
        ...
    } catch (IOException e){
        // do some recovery handling
    } finally {
        // close resources
    }
}
```

Gets executed in any case, at the end, always!

Handling Exceptions: Stop Propagation!



Java VM Runtime

`ReadTest.main();`

`ReadTest.main`

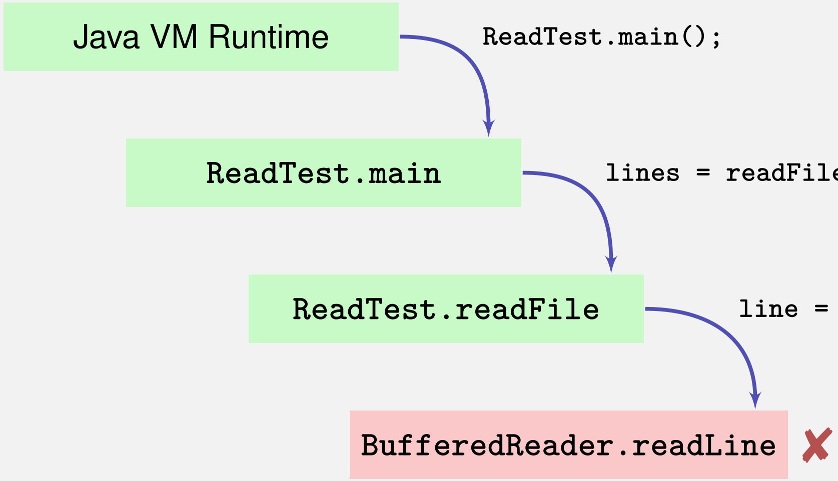
`lines = readFile("dataset.csv");`

`ReadTest.readFile`

`line = bufr.readLine();`

`BufferedReader.readLine`

Handling Exceptions: Stop Propagation!



Java VM Runtime

`ReadTest.main();`

`ReadTest.main`

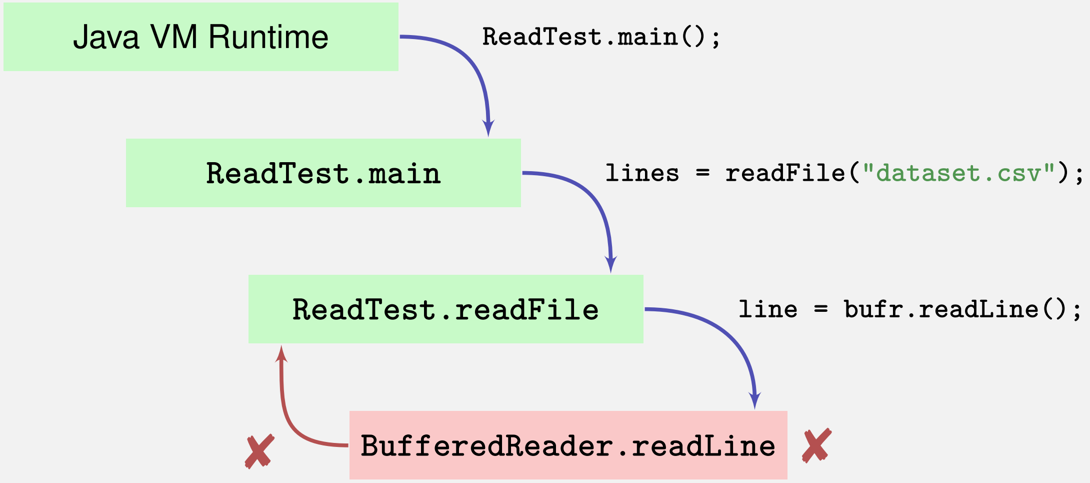
`lines = readFile("dataset.csv");`

`ReadTest.readFile`

`line = bufr.readLine();`

`BufferedReader.readLine` ✘

Handling Exceptions: Stop Propagation!



Java VM Runtime

`ReadTest.main();`

`ReadTest.main`

`lines = readFile("dataset.csv");`

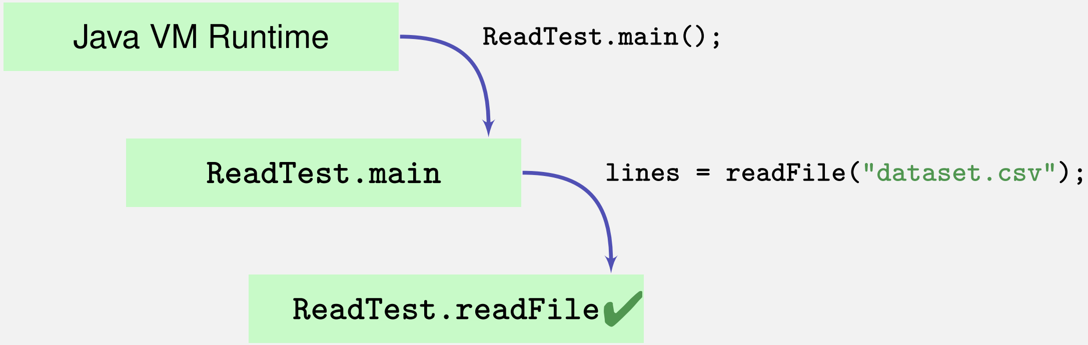
`ReadTest.readFile`

`line = bufr.readLine();`

`BufferedReader.readLine` ✗

✗

Handling Exceptions: Stop Propagation!



Java VM Runtime

`ReadTest.main();`

`ReadTest.main`

`lines = readFile("dataset.csv");`

`ReadTest.readFile` ✓

Exception caught!

Finally: Closing Resources

In Java, *resources* must be closed after use at all costs. Otherwise, memory won’t get freed.

Resources:

- Files
- Data streams
- UI elements
- . . .

Try-With-Resources Statement

Specific syntax to close resources *automatically*:

```
private static String[] readFile(String filename){
    try ( FileReader fr = new FileReader(filename);
         BufferedReader bufr = new BufferedReader(fr)) {
       ...
       line = bufr.readLine();
       ...
    } catch (IOException e){
       // do some recovery handling
    }
}
```

Try-With-Resources Statement

Specific syntax to close resources *automatically*:

```
private static String[] readFile(String filename){
    try ( FileReader fr = new FileReader(filename);
         BufferedReader bufr = new BufferedReader(fr)) {
       ...
       line = bufr.readLine();
       ...
    } catch (IOException e){
       // do some recovery handling
    }
}
```

Resources get opened here

Try-With-Resources Statement

Specific syntax to close resources *automatically*:

```java
private static String[] readFile(String filename){
    try ( FileReader fr = new FileReader(filename);
          BufferedReader bufr = new BufferedReader(fr)) {
        ...
        line = bufr.readLine();
        ...
    } catch (IOException e){
        // do some recovery handling
    }
}
```

Resources get closed automatically here

7. Functional Concepts in Java

Functional programming, lambda expressions, streams, pipelines

Functional vs. Imperative Programming

Imperative concepts

Functional Concepts

Functional vs. Imperative Programming

Imperative concepts

- Executing statements

Functional Concepts

- Evaluating expressions

Functional vs. Imperative Programming

Imperative concepts

- Executing statements
- State (e.g. Fields)

Functional Concepts

- Evaluating expressions
- Stateless

Functional vs. Imperative Programming

Imperative concepts

- Executing statements
- State (e.g. Fields)
- Mutable data types

Functional Concepts

- Evaluating expressions
- Stateless
- Immutable data types

Functional vs. Imperative Programming

Imperative concepts

- Executing statements
- State (e.g. Fields)
- Mutable data types
- Focus on data structures

Functional Concepts

- Evaluating expressions
- Stateless
- Immutable data types
- Focus on streams

Functional vs. Imperative Programming

Imperative concepts	Functional Concepts
Executing statements	Evaluating expressions
State (e.g. Fields)	Stateless
Mutable data types	Immutable data types
Focus on data structures	Focus on streams
Focus on “how”	Focus on “what”

Example: Reading of Files - Imperative

```
try (BufferedReader br=new BufferedReader(new FileReader("data.csv"))){
    LinkedList<Measurement> result = new LinkedList<>();
    br.readLine();
    String line ;
    while ((line = br.readLine()) != null){
        Measurement m = new Measurement(line);
        result .add(m);
    }
    return result ;
}
```

Example: Readong of Files - Functional

```
try (Stream<String> stream = Files.lines(Paths.get("data.csv"))) {




}
```

Example: Readong of Files - Functional

```
try (Stream<String> stream = Files.lines(Paths.get("data.csv"))) {

    return stream.skip(1).map(Measurement::new).collect(toList());


}
```

Streams

In Java, *Streams* are the basis for functional programming. Sources of streams:

- Files
- Arrays
- Data structures
- . . .

Streams

In Java, *Streams* are the basis for functional programming. Sources of streams:

- Files
- Arrays
- Data structures
- . . .

Example

```
Stream<String> stream = Files.lines (...))
```

Operations on Streams: Map

Map: Applying functions on individual elements of the stream

- Mathematical computations
- Creation of new objects based on existing elements.
- . . .

Operations on Streams: Map

Map: Applying functions on individual elements of the stream

- Mathematical computations
- Creation of new objects based on existing elements.
- . . .

Example

```
map(Measurement::new)
```

Operations on Streams: Reduce

Reduce: Aggregation of individual elements of a stream to one single value.

- Statistical aggregation
- Put elements in a data structure
- . . .

Operations on Streams: Reduce

Reduce: Aggregation of individual elements of a stream to one single value.

- Statistical aggregation
- Put elements in a data structure
- . . .

Example

```
collect (toList ())
```

Example: Search for Data - Imparative

```
List<Measurement> data = readCsvData();
Coordinate ref = readCoordinate();

for (Measurement m : data){
    if (m.position.near(ref)){
        System.out.println(m.originalLine);
    }
}
```

Example: Search for Data - Functional

```
List<Measurement> data = readCsvData();
Coordinate ref = readCoordinate();

data.stream()
          . filter (m −> ref.near(m.position))
          . forEach(System.out::println );
```

Operations on Streams: Filter

Filter: Filter individual elements of a stream.

- Remove illegal values
- Select values based on inquiries
- . . .

Operations on Streams: Filter

Filter: Filter individual elements of a stream.

- Remove illegal values
- Select values based on inquiries
- ...

Example

```
filter (m -> ref.near(m.position))
```

Operations on Streams: Side Effects

Sideeffects: The non-functional aspect: Execution on arbitrary operations based on individual elements.

- Input/Output
- Update data structures
- . . .

Operations on Streams: Side Effects

Sideeffects: The non-functional aspect: Execution on arbitrary operations based on individual elements.

- Input/Output
- Update data structures
- . . .

Example

```
forEach(System.out::println)
```

Functionality as Parameter

Operations on streams have *functionality* (code) as parameter, instead of *data*

Functionality as Parameter

Operations on streams have *functionality* (code) as parameter, instead of *data*

Possibility to pass functionality (instead of data)

- code snippets
- References on methods
- References to constructors

How can we do this?

Lambda Expressions

Lambda expressions are basically methods without names.

Normal method

```
double discriminant(double a, double b, double c){
    return b*b - 4*a*c;
}
```

Lambda Expressions

Lambda expressions are basically methods without names.

Normal method

```
double discriminant(double a, double b, double c){
    return b*b - 4*a*c;
}
```

Equivalent lambda expression

```
(double a, double b, double c) -> {
    return b*b - 4*a*c;
}
```

Lambda Expressions

Lambda expression

```
(double a, double b, double c) -> {
    return b*b - 4*a*c;
}
```

Lambda Expressions

Lambda expression

```
(double a, double b, double c) -> {
    return b*b - 4*a*c;
}
```

Without explicit type declaration of the parameters

```
(a, b, c) -> {
    return b*b - 4*a*c;
}
```

Lambda Expressions

Lambda expression

```
(double a, double b, double c) -> {
    return b*b - 4*a*c;
}
```

Without explicit type declaration of the parameters

```
(a, b, c) -> {
    return b*b - 4*a*c;
}
```

With a single expression instead of a block

```
(a, b, c) -> b*b - 4*a*c
```

Lambda Expression in the Example

Example

```
filter (m —> ref.near(m.position))
```

Lambda Expression in the Example

Example

```
filter (m -> ref.near(m.position))
```

- The method `filter` expects a method as parameter that takes a `Measurement` as parameter and returns a `boolean`.

Lambda Expression in the Example

Example

```
filter (m -> ref.near(m.position))
```

- The method `filter` expects a method as parameter that takes a `Measurement` as parameter and returns a `boolean`.
- `m` is a parameter of type `Measurement` ✓
- `ref.near(m.position)` is a single `boolean` expression ✓

Lambda Expression in the Example

Example

```
filter (m −> ref.near(m.position))
```

- The method `filter` expects a method as parameter that takes a `Measurement` as parameter and returns a `boolean`.
- `m` is a parameter of type `Measurement` ✓
- `ref.near(m.position)` is a single `boolean` expression ✓
- The variable `ref` from the defining context is accessible, if it is *effectively* constant (`final`).

References on Methods

To *call* a method on an object, we write:

```
object.method()
```

References on Methods

To *call* a method on an object, we write:

```
object.method()
```

To specify a *reference* to a method on an object, we write:

```
object::methode
```

References on Static Methods

To *call* a static method, we write:

```
Clazz.method()
```

References on Static Methods

To *call* a static method, we write:

```
Clazz.method()
```

To specify a *reference* to a static method, we write:

```
Clazz::method
```

Reference to a Method in the Example

Example

```
forEach(System.out::println)
```

Reference to a Method in the Example

Example

```
forEach(System.out::println)
```

- The method `forEach` expects a method, which doesn't return anything and takes an argument of type `Measurement`.

Reference to a Method in the Example

Example

```
forEach(System.out::println)
```

- The method `forEach` expects a method, which doesn't return anything and takes an argument of type `Measurement`.
- The method `println` on object `out` satisfies those properties ✓

References to Constructors

To *call* a constructor of a class, we write:

```
new Clazz()
```

References to Constructors

To *call* a constructor of a class, we write:

```
new Clazz()
```

To specify a *reference* to a constructor of a class, we write:

```
Clazz::new
```

References to a Constructor in the Example

Example

```
map(Measurement::new)
```

References to a Constructor in the Example

Example

```
map(Measurement::new)
```

- The method `map` expects a method that returns an object of a certain data types (it doesn't matter which) and an argument of type `String`.

References to a Constructor in the Example

Example

```
map(Measurement::new)
```

- The method `map` expects a method that returns an object of a certain data types (it doesn’t matter which) and an argument of type `String`.
- The constructor of the class `Measurement` satisfies this property ✓

Advantages and Disadvantages of Functional Programming

- Less error-prone
- Easier to maintain
- Allows for elegant programming constructs
- Independent on specific architecture

- Learn another language concept
- Details on the execution are unknown
- Super-imposed on an imperative language