Next Java lab: range objects

Examples

- `new Range(3)` is the immutable sequence of three elements 0, 1, 2
- `new Range(-2,2)` is the immutable sequence of four elements -2, -1, 0, 1
- `new Range(2,-2)` is the immutable empty sequence

Remark: all three ranges above can be implemented by using the same constant amount of memory

Ranges are iterable objects

For instance

```
for(int i : new Range(-2,2)){...}
```

is equivalent to

```
for(int i = -2; i < 2; i++){...}
```

Next Java lab: range objects

Implementation outline in Java

```java
public class Range implements Iterable<Integer> {

   // object fields ...

   // defines a range from start (inclusive) to end (exclusive)
   public Range(int start, int end) {...}

   // defines a range from 0 (inclusive) to end (exclusive)
   public Range(int end) {...}

   // implements the abstract method of Iterable, returns a new RangeIterator
   @Override
   public RangeIterator iterator() {...}
}

class RangeIterator implements Iterator<Integer> {

   // object fields and constructors ...

   @Override
   public boolean hasNext() {...}
   @Override
   public Integer next() {...}
}
```

Next Java lab: range objects

Demo

```
Range r = new Range(3); // interval between 0 (inclusive) and 3 (exclusive)
for (int x : r)
   for (int y : r)
      System.out.println(x + " " + y);
```

Prints

```
0 0
0 1
0 2
1 0
1 1
1 2
2 0
2 1
2 2
```

New useful Java features

Default methods and `var` declarations

- Local variable declarations with `var` (introduced with Java 10)
  local variables with no type declaration: type inferred by the compiler
- Default methods (introduced with Java 8)
  methods with a *default* body in interfaces

var declarations

Examples of **var** declarations

```
var r = new Range(2); // inferred type: Range
var it = r.iterator(); // inferred type: RangeIterator
var el = it.next(); // inferred type: Integer
// inferred type for s: HashSet<Integer>
var s = new HashSet<>(Arrays.asList(new Integer[] { 1, 2, 3, 4 }));
```

Main rules

- **var** only allowed for local variables (local variables are the variables declared in the bodies of constructors and methods)
- variables must be initialized, **null** not allowed
- no multiple variables, no array initializers

Default methods

Rules

- Interfaces can contain default object methods
- Default methods have a body, to define their default behavior
- Motivations:
  - code reuse
  - seamless code extension with new methods

Example of use: definition of *optional* methods

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    // optional method, by default it throws UnsupportedOperationException
    default void remove() {
        throw new UnsupportedOperationException();
    }
    ...
    var r = new Range(2);
    var it = r.iterator();
    it.next();
    it.remove(); // throws UnsupportedOperationException
}
```

Exceptions in Java

Motivation: better support for error handling

- the place where a failure occurs is often not the right point to handle it
- clear separation between
  - normal and abrupt execution
  - values and exceptions
    - values are computed only when execution completes normally
    - exceptions are thrown when there is a failure during the execution
- two separate constructs dedicated to exceptions:
  - error generation and propagation
  - error handling
- advantages: reliability
  - more effective way to debug code and detect bugs
  - force programmers to properly manage exceptional situations

Statement `throw`

Syntax and semantics

- syntax: **`throw`** *`e`*`;`
- static semantics: *e* must be an expression of type $T \leq$ `Throwable`
- dynamic semantics:
  - normal execution flow is interrupted, error is propagated to the callers and eventually handled or the program is terminated abruptly with a failure
  - if *e* evaluates to **`null`**, then `NullPointerException` is thrown

Java exceptions are objects of type `java.lang.Throwable`

Only instances of subtypes of `java.lang.Throwable` can be thrown

Statement `throw`

Examples

-
  ```
  throw new IOException("error message");
  ```

-
  ```
  Throwable cause;
  ...
  // exceptions can be chained
  // when an exception is the cause of another one
  throw new IOException("error message",cause);
  ```

-
  ```
  Throwable ex;
  ...
  throw ex;
  ```

-
  ```
  Throwable ex;
  ...
  throw ex.getCause();
  ```

Statement throw

Demo

```
public class TimerClass implements Timer {
    private int time = 60;
    private static void checkMinutes(int minutes) {
        if (minutes < 0 || minutes > 60)
            throw new IllegalArgumentException(); // line 9
    }
    public int reset(int minutes) {
        checkMinutes(minutes);                    // line 26
        var prevTime = time; // these lines might not be executed
        time = minutes * 60;
        return prevTime;
    }
    ...
}
public class ExceptionTest {
    public static void main(String[] args) {
        var timer = new TimerClass(30);
        timer.reset(-2);                          // line 7
        System.out.println(timer.getTime());      // not executed
    }
}
Exception in thread "main" java.lang.IllegalArgumentException
    at TimerClass.checkMinutes(TimerClass.java:9)
    at TimerClass.reset(TimerClass.java:26)
    at ExceptionTest.main(ExceptionTest.java:7)
```

Statement `throw`

Rules on error generation and propagation

- an exception originates from a **`throw`** statement in a method/constructor
- the thrown exception is propagated to the caller
- the caller can either handle the exception or propagate it to its caller
- if the exception propagates to the `main` method, and is not handled, then
  - the program terminates abruptly with a failure
  - information on the type of exception and the stack trace through which it has been propagated is printed out on the standard error stream

Statement **`try-catch`**

Exception handling

- exceptions are handled with the **`try-catch`** statement
- the **`try-catch`** statement may stop exception propagation

Example

```
public class ExceptionTest {
    public static void main(String[] args) {
        var timer = new TimerClass(30);
        try {
            timer.reset(Integer.parseInt(args[0]));
        } catch (Throwable e) { // unique handler for all types of exceptions
            timer.reset(0);
        }
        System.out.println(timer.getTime()); // the statement is always executed
    }
}
```

Motivation

Correct handling of exceptions avoid program crashes

Statement `try-catch`

Catch clauses can be multiple for different exceptions and bugs

```
try {
    ...
} catch (IOException e) {
    ... // code to recover error
} catch (Throwable e) { // unexpected exception, most likely a bug
        // bug reporting or logging required
    e.printStackTrace();
        // if error cannot be recovered, program should gracefully terminate
    ...
}
```

Rules

- catch clauses considered in left-to-right and top-to-bottom order
- subtyping is used to match clauses
- more specific exceptions must come first, static semantics forbids unreachable clauses (`Throwable` must always be the last clause)
- only one clause is used: the first that matches
- if no clause matches, then the caught exception is propagated

Statement try-catch

Demo: a complete example with multiple catch clauses

```
var mtch = Pattern.compile("[a-zA-Z][\\w]*").matcher("");
var group = 0;
// required args: args[0] a string, args[1] a reg-exp, args[2] an int
if (args.length < 3) {
   System.err.println("Error: Missing arguments");
   return;
}
mtch.reset(args[0]);
try {
   mtch.usePattern(Pattern.compile(args[1])); // may throw PatternSyntaxException
   group = Integer.parseInt(args[2]); // may throw NumberFormatException
} catch (PatternSyntaxException e) {
   System.out.println("Argument 2 is not a valid regular expression");
   System.out.println("Using default regular expression and group");
} catch (NumberFormatException e) {
   System.out.println("Argument 3 is not a valid integer, using default group");
}
if (group < 0 || group > mtch.groupCount()) {
   System.out.println("Argument 3 is not a valid group, using 0 as default");
   group = 0;
}
if (mtch.lookingAt())
   System.out.println("Matched string at group " + group + ": " + mtch.group(
        group));
```

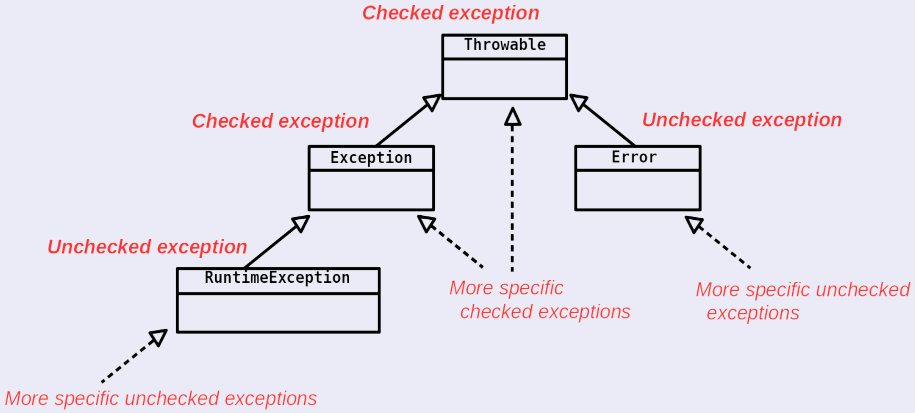
Unchecked and checked exceptions in Java

Exception classification

- Unchecked exceptions
  - errors: subclasses of `Error`
    serious problems (e.g. `OutOfMemoryError`, `StackOverflowError`)
  - runtime exceptions: subclasses of `RuntimeException`
    logic errors/precondition violations (e.g. `NullPointerException`, `IllegalArgumentException`)
- Checked exceptions: subclasses of `Exception` or `Throwable`
  Example: `java.io.IOException`
  In this case the user is forced to manage the exception in two ways:
  - either by handling the exception with **`try-catch`**
  - or by declaring that the constructor or method may throw the exception

Unchecked and checked exceptions in Java

Exception hierarchy



throws clauses

Exceptions can be declared in the headers of constructors and methods

Rules for checked exceptions

- Exception handling is enforced by the compiler for checked exceptions
- If the invocation of a constructor or method may throw a checked exception *E*, then
  - *E* is handled in the body with a **try**-**catch** (see `read1`)
  - or *E* is declared in the header (see `read2`)
- The static semantics forbids to catch a checked exception that can never be thrown

Example

```
static void read1(BufferedReader br) {
// does not throw or propagate checked exceptions
...
}
static void read2(BufferedReader br) throws IOException {
// could throw or propagate exceptions of type IOException
...
}
```

Error handling

The place where a failure occurs is often not the right point to handle it

Example 1: error handled as soon as possible

```
static void read1(BufferedReader br) {
    String line;
    do {
        try {
            line = br.readLine(); // may throw IOException
        } catch (IOException e) {
            System.err.println(e.getMessage());
            return;
        }
        if (line != null)
            System.out.println(line);
    } while (line != null); // if line == null then EOF has been reached
}
public void caller(BufferedReader br) {
    read1(br); // catching IOException here is a static error!
    ...
}
```

Error handling

The place where a failure occurs is often not the right point to handle it

Example 2: error better handled at an higher level

```
static void read2(BufferedReader br) throws IOException {
    String line;
    do {
        line = br.readLine(); // may throw IOException, 'throws' clause needed
        if (line != null)
            System.out.println(line);
    } while (line != null); // if line == null then EOF has been reached
}
public void caller(BufferedReader br) {
    try { // the caller has more control on method 'read'
        read2(br);
    } catch (IOException e) {
        System.err.println(e.getMessage());
        ... // asks the user another file to read
    }
    ...
}
```

Input/Output in Java

Main package `java.io`

- provides all basic features
- four parallel inheritance hierarchies:
  - input/output byte (binary) streams: `InputStream`, `OutputStream`
  - input/output char (text) streams: `java.lang.Readable` and `Reader`, `Writer`
- many classes implement the decorator design pattern to add extra features

More recent package `java.nio`

Other useful/advanced features

Decorator design pattern

In a nutshell

- a way to extend objects
- more flexible than inheritance: supports dynamic, multiple extensions of single objects
- a decorator wraps the object to be extended, and delegates to it the execution of some methods

Examples

- `BufferedReader` : constructor `BufferedReader(Reader)` allows buffering of characters of a `Reader` for efficiency
- `PushbackReader`: constructor `PushbackReader(Reader)` allows read characters of a `Reader` to be pushed back
- `PrintWriter`: constructor `PrintWriter(Writer)` allows formatted printing for a `Writer`

Convenient classes for input/output character streams

`java.io.BufferedReader`

- it is possible to read lines of characters with `readLine`
- it is only possible to decorate input character streams (type `Reader`)
- to decorate byte streams as `System.in`, decorator `InputStreamReader` must be created with constructor `InputStreamReader(InputStream in)`

  Example:

  ```
  new BufferedReader(new InputStreamReader(System.in))
  ```

`java.io.PrintWriter`

- it is possible to print lines of characters with `println`
- many variants of available constructors
  - `PrintWriter(String fileName)` to open files directly from their file name
  - `PrintWriter(Writer out)` to decorate character streams
  - `PrintWriter(OutputStream out)` to decorate byte streams

Input character streams

Example

```java
static BufferedReader tryOpen(String[] args) throws FileNotFoundException {
   if (args.length > 0) // tries to open textfile with name 'args[0]'
      return new BufferedReader(new FileReader(args[0]));
   // returns stdin decorated for a buffered char stream with default charset
   return new BufferedReader(new InputStreamReader(System.in));
}

static void read(BufferedReader br) throws IOException {
   // reads the lines of 'br' and prints them out on stdout
   String line;
   do {
      line = br.readLine(); // may throw IOException
      if (line != null)     // null means EOF
         System.out.println(line);
   } while (line != null);
}
```

try-catch-finally versus try-with-resources

try-catch-finally

a **`finally`** block is always executed at the end

Example with try-catch-finally

```
static void tryClose(Closeable c) {
   try {
      if (c != null) c.close(); // may throw IOException
   } catch (IOException e) { System.err.println(e.getMessage()); }
}
public static void main(String[] args) {
   BufferedReader br = null;
   try {
      br = tryOpen(args);        // may throw FileNotFoundException
      read(br);                  // may throw IOException
   } catch (IOException e) {     // FileNotFoundException ≤ IOException
      System.err.println(e.getMessage());
   } finally {                   // always executed
      tryClose(br);              // 'br' must be declared before try-catch-finally
   }
}
```

try-catch-finally versus try-with-resources

try-with-resources (since Java 8)

automatically closes “resources” and handle all possible exceptions

Example with try-with-resources

```
public static void main(String[] args) {
   try (var br = tryOpen(args)) { // br has type BufferedReader ≤ AutoCloseable
      read(br);                   // may throw IOException
   } catch (IOException e) {
      System.err.println(e.getMessage());
   }
}
```

Remarks

try-with-resources: simpler code, method `tryClose` not needed!

try-with-resources

Rules

- `try(...)` contains declarations of resources: local variables (as `bf`) declared and initialized, with scope extending as far as the try block
- the types of the resources must be subtypes of `AutoCloseable`
- resources are auto-closed (if non null) in the reverse order of initialization
- catch clauses manage also exceptions thrown during the initialization or automatic closing of resources