CS601: Principles of Software Development

Exceptions.

Olga A. Karpenko

Parts of this presentations are based on the Java Solftware Solutions book by Lewis&Loftus.

Announcements

- Lab 1 part 1 is due tonight at 11:59pm
  - Must be submitted to github
  - Passing provided tests is 80% of the grade
- Start working on Lab 1 part 2

What is an Exception?

- Event that disrupts the normal flow during execution
- Example: tried to open a file, but no such file exists

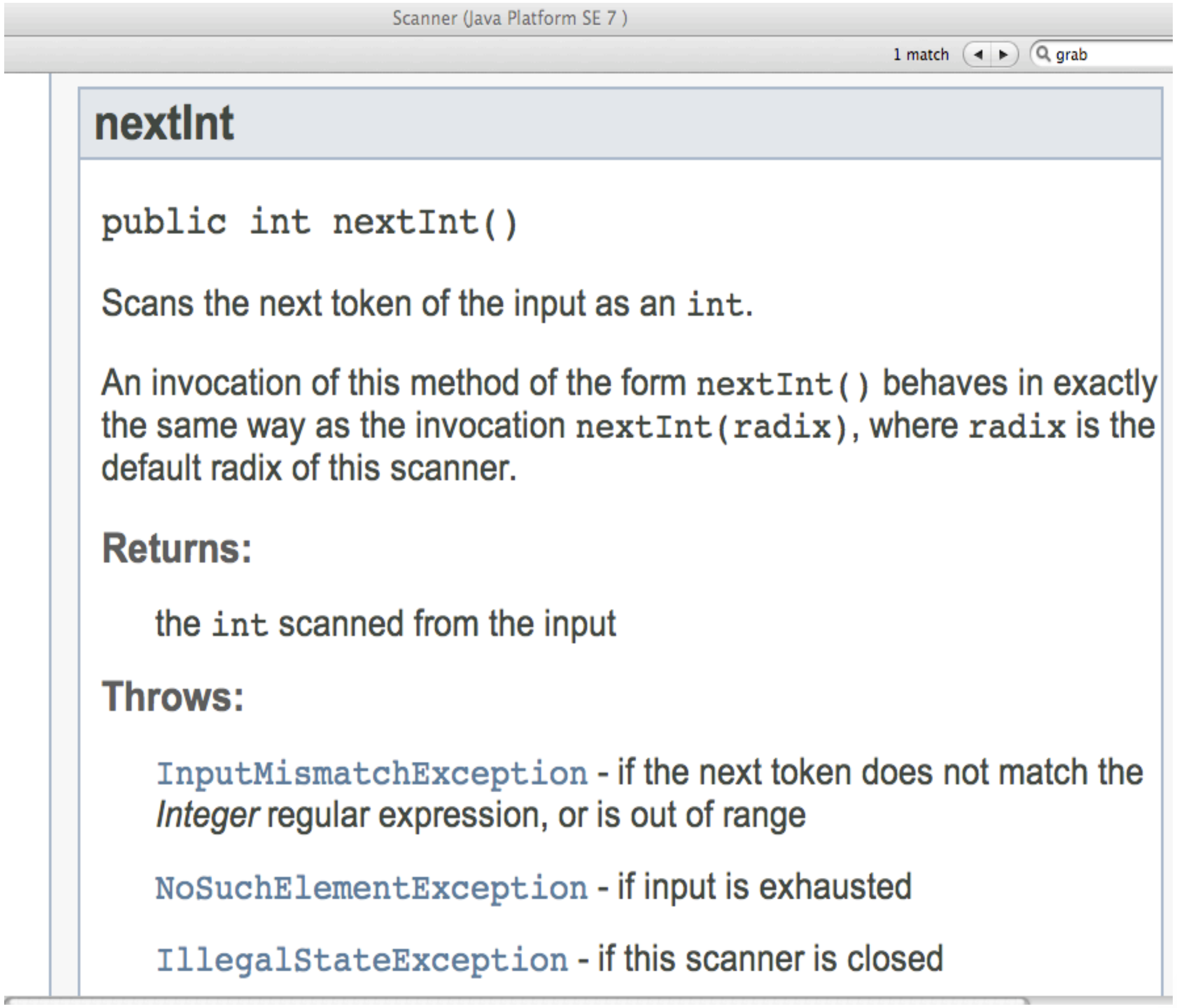
Exception Object

- Created by the method when exception occurs
- Contains information about what happened
  - type of exception
  - the state of the program when the error occurred

Exception: examples

- Array Index Out of Bounds
- File Not Found
- NullPointerException
- …

Exceptions and the API



nextInt

```
public int nextInt()
```

Scans the next token of the input as an `int`.

An invocation of this method of the form `nextInt()` behaves in exactly the same way as the invocation `nextInt(radix)`, where `radix` is the default radix of this scanner.

Returns:

the `int` scanned from the input

Throws:

`InputMismatchException` - if the next token does not match the *Integer* regular expression, or is out of range

`NoSuchElementException` - if input is exhausted

`IllegalStateException` - if this scanner is closed

Exceptions

- Exceptions are *thrown* by some statements
- May be *caught* and *handled* by another piece of code
- In Java: a predefined set of exceptions that can occur during execution

Java Exception Handling

- Enable program to operate even in the presence of an exception
  - Note problem and continue
  - Terminate gracefully
- Important for building robust software

Exception Handling

- When exception occurs, a program can
  - ignore it
  - handle it where it occurs
  - handle it an another place in the program
- How to handle each exception is an important design decision

Ignoring Exceptions

- If an exception is ignored -> the program will **crash** and print the error message
- The message includes a *call stack trace* that:
  - indicates the line on which the exception occurred
  - shows the method call trail

Catching Exceptions

- The **try/catch** statement :

```
try {
   //statements that may throw an exception
}
catch (Exception_type name) {
   //do something
}
finally { // optional
   //code that will execute whether or not
an exception is thrown
}
```

Execution Flow

- No exceptions -> the catch block is skipped
- Exception was thrown -> goes to the catch block
- In either case, statements in the finally block will be executed

try Statement: Example

```java
int a = 5;
int b = 0;
try {
   int c = a / b;
   System.out.println(c);
}
catch (ArithmeticException e) {
   System.out.println("Can't divide by 0.");
}
```

- See `ArithmeticExceptionDemo.java`

The finally Clause

- Optional
- Is always executed
- No exception -> finally clause is executed after the statements in the try block
- Exception –>finally clause is executed after the statements in the appropriate catch clause

Example

```
BufferedReader reader = null;
try {
     reader = Files.newBufferedReader(path, 
Charset.forName("UTF-8"));
     String line = null;
     while ((line = reader.readLine()) != null) {
            System.out.println(line);
     }
}
catch (IOException e) {
    System.out.println(e);
}
finally {
    // Close the reader here..
}
```

Quick Check

What will happen if exception of type 2 occurs in statement 1 ?

```
try {
      statement1;
      statement2;
}
catch (ExceptionType1 e) {
    // statements executed when
    // exception is thrown
}
finally {
    // statements that are executed
    // in any case
      statement3;
}
statement4;
```

try Statement

- There can be more than one catch block
- Catch each type of exception in a separate catch block

Example: Two catch blocks

```
BufferedReader br;
try {
     FileReader f = new FileReader("file");
     br = new BufferedReader(f);
     String line = br.readLine();
     System.out.println(line);
}
catch (IOException e) {
     System.out.println("Error reading file");
}
catch(Exception e) {
    System.out.println("Exception occurred");
} // need to close br in the finally block
```

Order of Catch Blocks

- Order Matters!
- A catch block of a subclass should appear before the catch block of a superclass

Order of Catch Blocks

```
try {
  …
}
catch(Exception ex) {
...
}
catch(IOException ex) {
…
}
```

Order of Catch Blocks

```
try {
 …
}
catch(IOException ex) {
...
}
catch(Exception ex) {
…
}
```

- IOException is a subclass of Exception

try Statement

- Starting Java 7, several exceptions can be intercepted in one catch block

Example

```
try {
    // statements that might throw
    // PrintException or IOException
}
catch (PrintException | IOException e) {
    System.out.println(e.getMessage());
}
```

From "Java for Dummies Quick Reference" by Doug Lowe

try-with-resources

- Starting Java 7
- A try statement that declares one or more resources
  - Resource: an object that must be closed after the program is finished with it. Example: file
- Ensures that each resource is closed at the end of the statement

try-with-resources

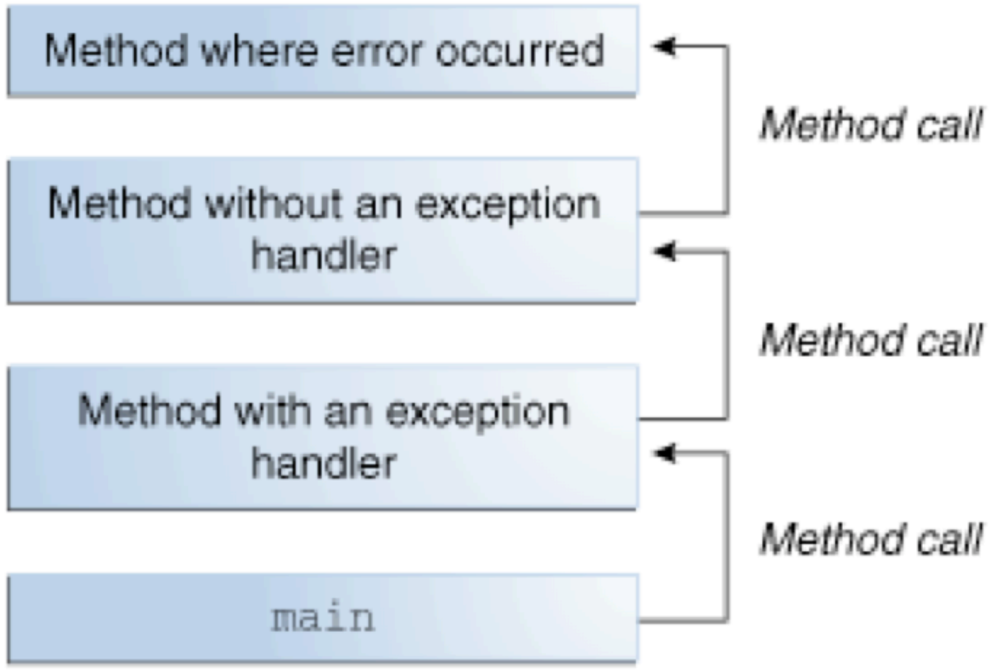
```
try (BufferedReader reader = Files.newBufferedReader(path,
Charset.forName("UTF-8"))) {
    String line = null;
    while ((line = reader.readLine()) != null) {
        System.out.println(line);
    }
}
catch (IOException e) {
    System.out.println(e);
}
```

- The resource in this example is BufferedReader
  - will be closed automatically
  - regardless of whether exception occurs or not

Exception Propagation

- An exception can be handled at a higher level
  - if it is not appropriate to handle it where it occurs
- Exceptions *propagate* up through the method calling hierarchy
  - until they are caught and handled or
  - until they reach the level of the main method

Call Stack


Method where error occurred
Method call
Method without an exception handler
Method call
Method with an exception handler
Method call
main

http://docs.oracle.com/javase/tutorial/essential/exceptions/definition.html

Call Stack

- Stores information about methods that are executing
- Contains *stack frames*
- Can push and pop stack frames

Stack Frame

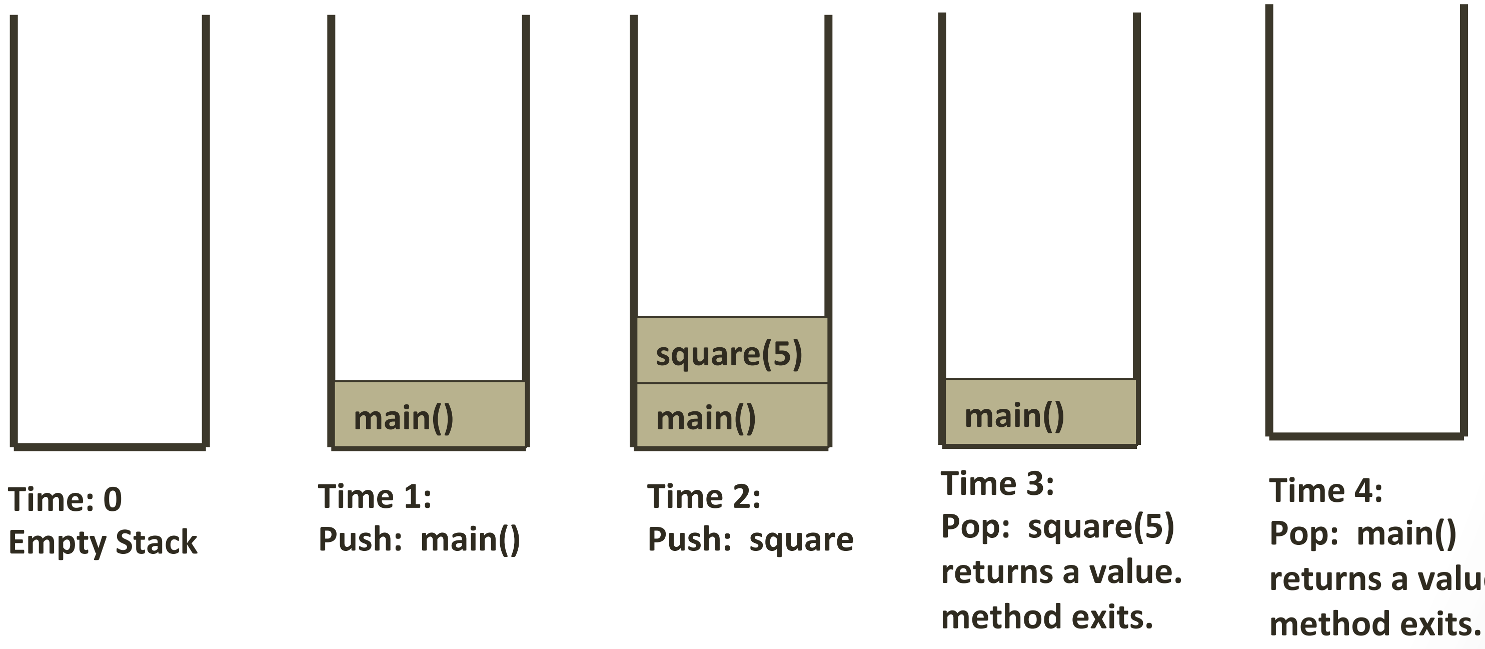
- Stores return address, local variables and parameters
- Is pushed onto the call stack
  - When the method is called

Stack Frame

- When the method returns, we pop the stack frame
  - Control goes to the method that called it
  - Execution starts from point immediately after the recursive call

Call Stack

- Assume main() calls the method square(5)



The slide is courtesy of Evan Korth, NYU

Exception Handler

- The call stack is searched for a method that can handle the exception
  - Starts with the method where the error occurred
  - Proceeds through the call stack in reverse order
- When a handler is found, the runtime system passes the exception to the handler
- Found no appropriate exception handler -> the program crashes

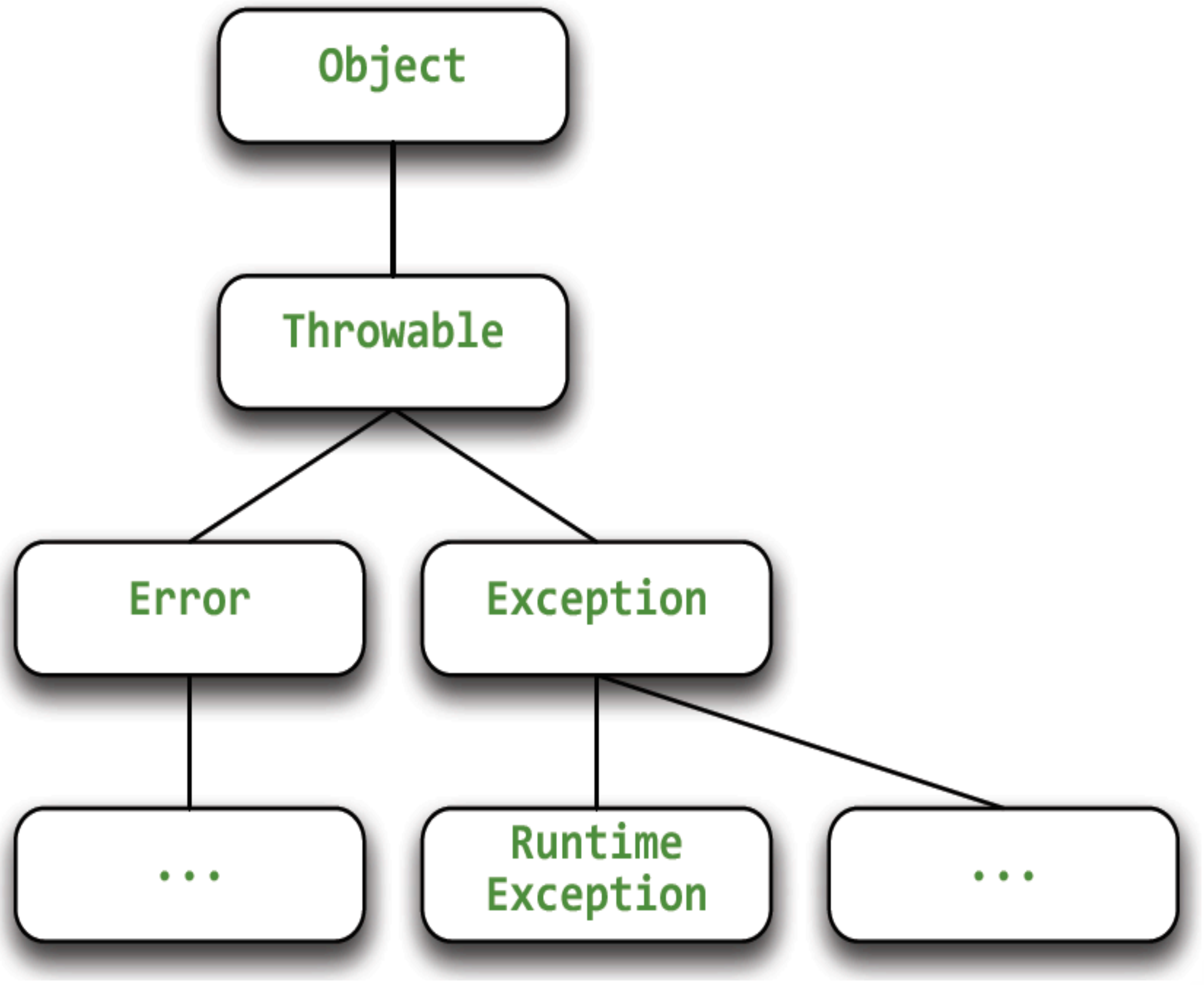
Exception Propagation

- See `Propagation.java`
- See `ExceptionScope.java`

The Exception Class Hierarchy

- Exception classes are related by inheritance
- All error and exception classes are descendents of the `Throwable` class

Exception Hierarchy



Checked vs Unchecked Exceptions

Checked Exceptions

- Should be anticipated by programmer
  - – e.g. unable to open a file

  Example: File input/output exceptions
- Must "deal" these exceptions in the program
  - Enforced by compiler
  - Forces the programmer to think how to handle the exceptional situation

The Catch or Specify Requirement

- For code that can throw checked exceptions, do one of the following
  - Catch with try/catch statement
  - List the exception in the throws clause of any method that may throw or propagate it
- Otherwise won't compile

throws Keyword

```
public void readLine() throws IOException {

     FileReader f = new FileReader(“file.txt”);
     BufferedReader br = new BufferedReader(f);
     String line = br.readLine();
     System.out.println(line);
     br.close();
}
```

- Telling Java that this code can throw IOException
- Hopefully one of the methods below readLine in the Call Stack will handle it

throws Keyword

```
public static void main(String[] args) {
    try {
        readSingleLine();
    }
    catch (IOException e) {
        System.out.println(e.getMessage());
    }
}
```

Unchecked Exceptions

- Any classes under `RuntimeException` and `Error`
- Can be handled, but not required
  - e.g. divide by zero
- Often indicates code defects/bugs
  - e.g. accessing array out of bounds
    - Often better to fix the bug rather than catching the exception

Ignoring the exception

- Also called “swallowing the exception”
- The catch block contains no statements:

```
try {
    // Statements that might throw
    // FileNotFoundException
}
catch (FileNotFoundException e) {
}
```

Ignoring the exception: Wrong

- Bad programming practice
  - Information about the exception is lost forever
  - Program errors may go undetected
- Instead, handle or re-throw the exception

```
try {
    // Statements that might throw
    // FileNotFoundException
}
catch (FileNotFoundException e) {
}
```

Custom Exceptions

- Throw the custom exception when something goes wrong
- Example (see `InsufficientFundsException, BankAccount`)
  1. Write a class `InsufficientFundsException` that extends class Exception
  2. In class BankAccount you might want to call:

```
public void withdraw(double amount) throws
InsufficientFundsException {
      if (amount > balance)
         throw new InsufficientFundsException(amount,
           "Not enough funds in your account.");
      this.balance -= amount;
}
```

Wrapping the exception

- Wrap and throw another exception

```
catch (NoSuchMethodException e) {
  throw new MyServiceException("Couldn't
process request", e);
}
```

How to handle a given exception?

- No single rule that tells you when
  - to catch/re-throw
  - when to use checked /unchecked exceptions
- Generally, re-throw an exception to the layer where you can handle it

References

- Exception Handling Anti-patterns:
  - https://community.oracle.com/docs/DOC-983543