

Programming 1

Lecture 7 – Exception, Recursion, Overloading, Scopes

Up next...

Exception handling

Errors in Java

- How to deal with errors?
 - Terminate program
 - Try an alternative solution
- In Java, errors are delivered as **Exceptions**.
- Exception handling provides a flexible mechanism for reporting and dealing with errors.

Error handling in old languages

- For non-critical errors: display a message and continue execution.
 - e.g. warnings, notices...
- For critical (fatal) errors: display a message and terminate program.
 - e.g. array out of bound, division by zero...

Error handling in Java

- In case of an unexpected situation, throw a suitable Exception.
 - e.g. invalid data, unable to read file, lost connection...
- Detect Exception with **try...catch**.
- Write code to deal with the Exception.
 - a.k.a. perform alternative actions.
- Program should never terminate unexpectedly!

Java Exceptions

- Event that occurs when something *unexpected* happens

```
int[] a = {1,2,3}; // a[0], a[1], a[2]  
int b = a[a.length]; // IndexOutOfBoundsException
```

```
String s; // s is null  
char c = s.charAt(i); // NullPointerException
```

Why use an Exception?

- To tell the code using your method that something went wrong

How do exceptions “happen”?

- Java doesn't know what to do, so it
 - Creates an Exception object
 - Includes some useful information
 - “throws” the Exception

Using Exception

- `Exception` is a special data type (just like `String`) which represents errors in Java (and many other languages).

```
Exception e = new Exception("404 Not Found!");  
System.out.println(e.getMessage());  
throw e; // intentionally raise an error
```

- However, `Exception` is meant to be “thrown”.

Using Exception

- It is used to indicate errors.
- It is used when there's some problem. E.g.
 - Data error
 - Inappropriate action from user
 - Something is not found or missing (data, library)
- Exceptions are meant to be handled, so that a program can response to problematic situations.
 - a.k.a. Even when some data is missing, the app keeps running smoothly (with some "Plan B" when an Exception is caught).

```
int n = sc.nextInt();  
if (n < 0) {  
    throw new Exception("Negative number entered");  
}
```

Using Exception

- It is `Throwable` (we can `throw` it)
- An uncaught Exception stops the program.
 - a.k.a. A caught one does not.

run:

```
Exception in thread "main" java.lang.Exception: Negative number entered  
    at Example9.main(Example9.java:16)  
C:\Users\fairy25\AppData\Local\NetBeans\Cache\8.1\executor-snippets\run.xml:53: Java returned: 1  
BUILD FAILED (total time: 0 seconds)
```

So how do I catch an Exception?

try-catch syntax

try block

Statement(s) that may throw **Exception**

```
try {  
    int a = sc.nextInt();  
} catch (Exception e) {  
    System.out.println("Input error!");  
}
```

The type of **Exception** that needs to be caught

catch block

Statement(s) that execute when an **Exception** is thrown (error occurred)

try-catch example

```
try {  
    int a = sc.nextInt();  
} catch (InputMismatchException e) {  
    System.out.println("That's not an integer!");  
    System.out.println("Try again: ");  
    sc.nextLine(); // clear newline char  
    int a = sc.nextInt();  
}
```

Output:

asd

That's not an integer!

Try again:

15

BUILD SUCCESSFUL (total time: 5 seconds)

Exception w/out try...catch

```
int n;  
System.out.print("Enter an integer: ");  
n = sc.nextInt();
```

run:

Enter an integer: asds

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 Example9.main([Example9.java:20](#))

[C:\Users\fairy25\AppData\Local\NetBeans\Cache\8.1\executor-snippets\run.xml:53](#): Java returned: 1

BUILD FAILED (total time: 1 second)

Exception with try...catch

```
int n;  
boolean gotIt = false;  
while (!gotIt) {  
    try {  
        System.out.print("Enter an integer: ");  
        n = sc.nextInt();  
        gotIt = true;  
    } catch (Exception e) {  
        System.err.print("Nah, don't try to fool me!");  
        sc.nextLine(); // try commenting out this line  
    }  
}
```

run:

Enter an integer: asd

Nah, don't try to fool me!

Enter an integer: 3

BUILD SUCCESSFUL (total time: 7 seconds)

Up next...

Recursion

Calling a method recursively

- **Recursion:** calling a method from inside itself.
- See example: calculating $x!$

```
public static int factorial(int x) throws Exception {  
    if (x < 1) {  
        throw new Exception("Invalid input!");  
    } else if (x == 1) {  
        return 1;  
    } else {  
        return x * factorial(x - 1);  
    }  
}
```


Calling a method recursively

- Consider: `factorial(5)`

Method call	Return expression	Value (bottom up)
<code>factorial(5)</code>	<code>5 * factorial(4)</code>	120
<code>factorial(4)</code>	<code>4 * factorial(3)</code>	24
<code>factorial(3)</code>	<code>3 * factorial(2)</code>	6
<code>factorial(2)</code>	<code>2 * factorial(1)</code>	2
<code>factorial(1)</code>	1	1

Recursion example: GCD

$$\text{gcd}(a, 0) = a$$

$$\text{gcd}(a, b) = \text{gcd}(b, a \bmod b)$$

```
public static int gcd(int a, int b) {  
    if (b == 0) {  
        return a;  
    } else {  
        return gcd(b, a % b);  
    }  
}
```

Up next...

Method overloading

Method overloads

- **Definition:** Two or more methods by the same name but different parameters.

```
public class Math {  
    public static long abs(long x) {  
        if (x < 0) return -x;  
        return x;  
    }  
  
    public static float abs(float x) {  
        if (x < 0) return -x;  
        return x;  
    }  
}
```

How overloading works

- When we call a method, the compiler must determine which of the methods to use through a process called binding.
 - Binding is matching a method's signature to how it is called.
 - A method's signature consists of its name and the data types of its parameters.
- Signature of the methods in the previous example:
 - `abs(long)`
 - `abs(float)`

How overloading works

- We cannot have methods with the same name and same list of parameters, EVEN IF THEY HAVE A DIFFERENT return type.
- Example of invalid overloading methods:

```
public double add(int a, int b) {...}
```

```
public int add(int x, int y) {...}
```

Calling method overloads

- A suitable overload is automatically selected.
- Automatic type conversion is applied.
 - E.g. an overload with `long` parameter is selected for `int` input
 - E.g. **no suitable overload** is found because `double` cannot be automatically converted to `long` or `float`

```
public static void main(String[] args) {  
    long a = -500;  
    System.out.println(abs(a)); // abs (long x) is called  
    int b = 10;  
    System.out.println(abs(b)); // abs (long x)  
    float c = 5.1f;  
    System.out.println(abs(c)); // abs (float x)  
    double d = 300;  
    System.out.println(abs(d)); // no suitable method  
}
```

Constructor overloading

- One of the more useful uses of method overloading is to overload constructors.
- More options when creating new objects.
- We can have many constructors which
 - Create empty objects
 - Create partial objects
 - Or, create complete objects

Packages

- All of the Java classes are organized into packages.
 - A package is a group of related classes.
- A class has a path that shows which package it belongs to:
 - `java.lang.String`, `java.lang.Math`, `java.util.Scanner`...
- Classes in `java.lang` package are available for use without importing. Others need importing.
 - `import java.util.Scanner;`
- All classes in a package can be imported using wildcard:
 - `import java.util.*;`

Up next...

Variable scopes

Variable scopes

- **Definition:** The scope of a variable is the part of the program that can refer to that variable by name.
- Three levels of Java variable scopes:
 - Class (global) scope
 - Declared outside of methods
 - Method scope
 - Declared inside a method, but not inside a block
 - Block scope
 - E.g. if...else, for, while, do, switch, try...catch

The **global** scope

- A class member refers to one of the following:
 - A **static** method
 - A **static** variable
(declared within the class and outside of methods)
- Class member vs. instance member:
 - A **static** member belongs to the class.
 - A **non-static** member does not belong to the class, but to an **instance** of that class.

Class members (**static**)

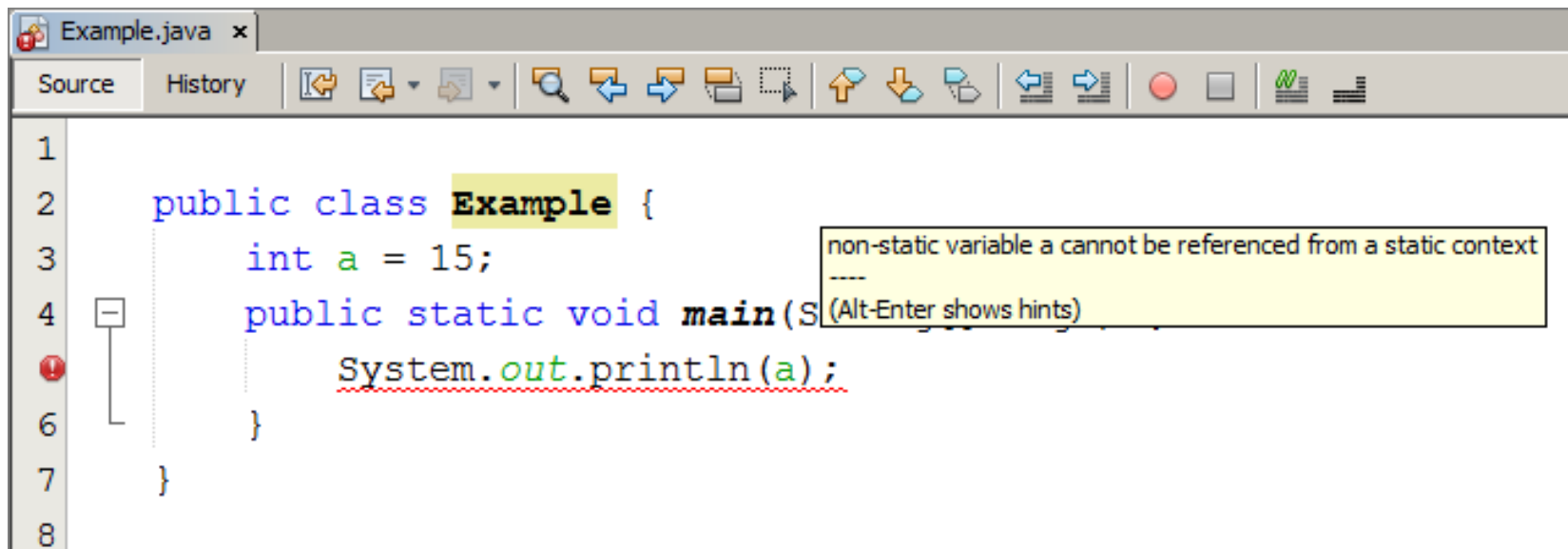
```
public class Example {  
  
    static int MAX_WIDTH = 1024;  
  
    public static int distance(int x, int y) {  
        return Math.abs(y - x);  
    }  
}
```

Instance members (**non-static**)

```
public class Example {  
  
    int size = 100;  
  
    public int distance(int x, int y) {  
        return Math.abs(y - x);  
    }  
}
```

The **global** scope

- The **main** method is **static**.
- **non-static** members (variables, methods) cannot be accessed from the **main** method or any other **static** method.



The screenshot shows an IDE window titled "Example.java" with a toolbar and a code editor. The code editor contains the following Java code:

```
1  
2 public class Example {  
3     int a = 15;  
4     public static void main(S  
5         System.out.println(a);  
6     }  
7 }  
8
```

A red error icon is visible on line 5. A tooltip box is displayed over the error, containing the text: "non-static variable a cannot be referenced from a static context" and "(Alt-Enter shows hints)".

The `global` scope

- A global variable is declared inside class and outside any method.
- `non-static` variables can only be accessed in `non-static` methods.
- `static` variables can be accessed anywhere in `static` and `non-static` methods.
 - We use `static` global variables in this course.

The **global** scope

- Example use of a global variable:

```
public class Example {  
  
    static int SIZE = 100;  
  
    public static void main(String[] args) {  
        int[] numbers = new int[SIZE];  
    }  
}
```


The **global** scope

- Another example use of **global** variables:
(which can be accessed by all methods in a class)

```
public static int a = 5; // a global variable
public static void firstMethod() {
    int b = a + 3;
}
public static void secondMethod() {
    System.out.println(a); // ok
}
```

- A **global** variable and a **method** are both class members. They are on the same level (like siblings).

The **local** scope

- **Definition:** Local variables, or method-level variables, are declared inside a method and outside any block.

```
public class Example {  
    static int globalNum = 15;  
    public static void main(String[] args) {  
        int localNum = globalNum - 4;  
        System.out.println(localNum);  
    }  
}
```

The **local** scope

- Variables declared inside a method are **local** to the method.
- These variables can't be accessed outside the method (a.k.a in other methods).

```
public static void firstMethod() {  
    int a = 5;  
}  
public static void secondMethod() {  
    System.out.println(a); // error: cannot find symbol  
}
```

The **block** scope

- **Definition:** A variable declared inside pair of brackets **{** and **}** (a block) has scope within the brackets only.
- This scope applies to:
 - **if...else** statements
 - Loops
 - **try...catch** statements
 - **switch**
 - Generic code blocks.

The **block** scope

- Example: **if...else** statements

```
if (pw.length() >= 8) {  
    int score = 1;  
} else {  
    int score = 0;  
}  
System.out.println(score); // error: cannot find symbol
```

- Example: **for** loop

```
for (int i = 0; i < 5; i++) {  
    System.out.println("i = " + i);  
}  
System.out.println("Now, i = " + i); // error
```

The **block** scope

- Example: **while** loop

```
int i = 0;
while (i < 10) {
    int j = 2;
    i = i + j;
}
System.out.println(j); // error: cannot find symbol
```

- Example: **switch** statement

```
int option = 1;
switch (option) {
    case 1:
        int a = 123;
}
System.out.println(a); // error
```

The **block** scope

- Example: generic code block

```
int a = 1;
{
    int n = 20;
    a = a + n;
}
System.out.println(a); // ok, prints 21
System.out.println(n); // error: cannot find symbol
```