Lecture Set #7: Exceptions & Mutability Issues

1. Break and Continue for Loops
2. Exceptions
3. Mutability/Immutability
4. StringBuffer class
“this” – when in an instance method: always represents the current object

optional to indicate an instance method or an instance data member is of the current object
when you have a local variable or a parameter named the same as a instance data member of the same class, use “this” to override the default when you need to pass the whole current object as the argument to another method
when you are writing a constructor and want to call a different constructor to construct the current object
break from loops

break can also be used to exit immediately from any loop

while
do-while
for

e.g. “Read numbers from input until negative number encountered”
  - Scanner sc = new Scanner (System.in);
  - int n;
  - while (true) {
  -     n = sc.nextInt ();
  -     if (n < 0) {
  -         break;
  -     } else {
  -         <process n>;
  -     }
  - }

Loop only terminates when break executed

This only happens when n < 0

“breaks past” if statements

Always breaks to first enclosing loop
Warning about break

Undisciplined use of break can make loops impossible to understand
   Termination of loops without break can be understood purely by looking while, for parts
   When break included, arbitrary termination behavior can be introduced

Rule of thumb: use break only when loop condition is always true (i.e. break is only way to terminate loop)
When you use it, make sure it has a good comment explaining what is happening
continue Statement

continue can also be used to affect loops
break halts loops
continue jumps to bottom of loop body

Following prints even numbers between 0 and 10
  • for (int i = 0; i <= 10; i++){
    • if (i % 2 == 1) {
    •       continue;
    •   }
    •   System.out.println (i);
    • }

Effect of continue statement is to jump to bottom of loop immediately when i is odd
This bypasses println!

continue should be avoided
Confusing
Easy equivalents exist (e.g. if-else)
Included in Java mainly for historical reasons

When you use it, make sure it has a good comment explaining what is happening
Exceptions

Programs can generate errors

Arithmetic

Divide by zero, overflows, …

Object / Array

Using a null reference, illegal array index, …

File and I/O

Nonexistent file, attempt to read past the end of the file, (we’ll see more about file I/O later in course), …

Application-specific

Errors particular to application (e.g., attempt to remove a nonexistent customer from a database)

In Java: something that is outside the norm = exception

What to do when an error occurs?

1. Basically ignore it: Print an error message and terminate?
2. Have the method handle it internally: Handle error in the code where the problem lies as best you can.
3. Have the method pass it off to someone else to handle: Return “error code” so that whoever called this function can handle it.
4. Modern language approach: Cause “exception” to be thrown (and caught (or processed) by any function up the stack trace)
Exception Behavior

If program generates ("throws") exception then default behavior is:
Java clobbers ("aborts") the program

Stack trace is printed showing where exception was generated (red and blue in Eclipse window)

Example

```java
public int mpg(int miles, int gallons) {
    return miles/gallons;
}
```

Throws an exception and terminates the program.
Throwing Exceptions Yourself

To throw an exception, use throw command:

```java
throw e;
```

`e` must evaluate to an exception object

You can create exceptions just like other objects, e.g.:

```java
RuntimeException e = new RuntimeException("Uh oh");
```

`RuntimeException` is a class

Calling `new` this way invokes constructor for this class

`RuntimeException` generalizes other kinds of exceptions (e.g. `ArithmeticException`)
Exceptions, Classes and Types

Exceptions are objects.
Some examples from the Java class library (mostly java.lang):

- **ArithmeticException**: Used e.g. for divide by zero
- **NullPointerException**: attempt to access an object with a null reference
- **IndexOutOfBoundsException**: array or string index out of range
- **ArrayStoreException**: attempting to store wrong type of object in array
- **EmptyStackException**: attempt to pop an empty Stack (java.util)
- **IOException**: attempt to perform an illegal input/output operation (java.io)
- **NumberFormatException**: attempt to convert an invalid string into a number (e.g., when calling Integer.parseInt())
- **RuntimeException**: general run-time error (subsumes above)
- **Exception**: The most generic type of exception
public int mpg(int miles, int gallons) {
    if (gallons == 0) {
        throw new NullPointerException();
    } else {
        return miles/gallons;
    }
}
Java Exceptions in Detail

Exceptions are (special) objects in Java
They are created from classes
The classes are derived ("inherit") from a special class, Throwable
We will learn more about inheritance, etc., later

Every exception object / class has:
Exception(String message)
  • Constructor taking an explanation as an argument
String getMessage()
  • Method returning the embedded message of the exception
void printStackTrace()
  • Method printing the call stack when the exception was thrown
Handling Exceptions

Aborting program not always a good idea
E-mail: can’t lose messages
E-commerce: must ensure correct handling of private info in case of crash
Antilock braking, air-traffic control: must recover and keep working

Java provides the programmer with mechanisms for recovering from exceptions
Java Exception Terminology

When an anomaly is detected during program execution, the JVM throws a particular type of exception. There are built-in exceptions. Users can also define their own (more later).

To avoid crashing, a program can catch a thrown exception (if it isn’t caught – you see the red and blue messages – stack trace). An exception generated by a piece of code can only be caught if the program is alerted. This process is called trying the piece of code.
Catch Example

try {
    System.out.println("Start");
    mpg(5, 0);
    System.out.println("Finish");
} catch (Exception e) {
    System.out.println("e = " + e);
}
Exception Propagation

Goes out to caller if not handled:
Exception thrown in one method ...

... but caught in another

Java uses exception propagation to look for exception handlers

When an exception occurs, Java pops back up the call stack to each of the calling methods to see whether the exception is being handled (by a try-catch block). This is exception propagation

The first method it finds that catches the exception will have its catch block executed. Execution resumes normally in the method after this catch block

If we get all the way back to main and no method catches this exception, Java catches it and aborts your program
Finally Block

Always run:
  When no exception has been thrown

  When an exception has been thrown but the exception was handled before that point in time

  When an exception has been thrown but the exception was NOT handled before that point in time

finally = ALWAYS
Exception Handling: Example

DateReader.java
Prompts user for a date in mm/dd/yyyy format
Prints year

Program uses:
substring method
  • May throw IndexOutOfBoundsException
Integer.parseInt method
  • May throw NumberFormatException
getYear method (if d is null)
  • May throw NullPointerException

How do we know about these exceptions? Javadoc!
http://java.sun.com/j2se/1.5.0/docs/api/java/lang/package-summary.html
What about Strings and Aliasing?

String objects are *immutable*; fields cannot be changed once created

Mutable objects: fields (values of instance variables) can be changed by a call to some function (e.g. Cat, Student, etc.)

Immutable objects: fields (values of instance variables) cannot be changed by any call to any function

See String API:
http://java.sun.com/j2se/1.3/docs/api/java/lang/package-summary.html

In the Cat and CatOwner example:
when one object is assigned to another, an alias is created

Cat a = new Cat(“Fluffy”);
Cat b = a;
Which picture represents the current status of memory?

Stack

Heap

Stack

Heap

"Fred"
Mutable Strings

Strings are immutable
   Once a String object is created, it cannot be altered

Sometimes mutable strings would be handy
   Sometimes a small change needs to be made to a string (e.g. misspelled name)
     Don’t want to create a whole new String object in this case

StringBuffer: Java’s class for mutable Strings
StringBuffer Basics

See documentation at:
http://java.sun.com/j2se/1.5.0/docs/api/java/lang/StringBuffer.html

Main methods
append: add characters to end
insert: add characters in middle
delete: remove characters

Note
append, insert return object of type StringBuffer
This is alias to object that the methods belong to!

See StringBufferExample.java