ECS 160 – Discussion

Java Basics

Instructor: Tapti Palit

Teaching Assistant: **Xingming Xu**

UCDAVIS

Agenda

- **TA introduction**
- Exceptions
- Generics
- Collections
- Boxing/Unboxing
- File I/O

TA details

- Name: Xingming Xu
- Background:
  - MSCS student
  - BCS @ UWaterloo
- Email: xmxu@ucdavis.edu
- Office hours: 10-11AM TR @ Kemper Hall, 55 (in the basement)

Agenda

- TA introduction
- **Exceptions**
- Generics
- Collections
- Boxing/Unboxing
- File I/O

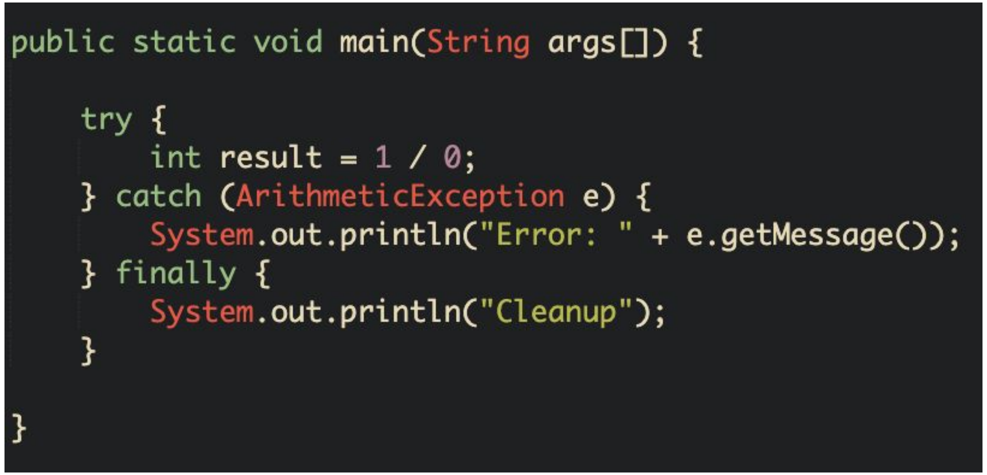
UCDAVIS

Exceptions

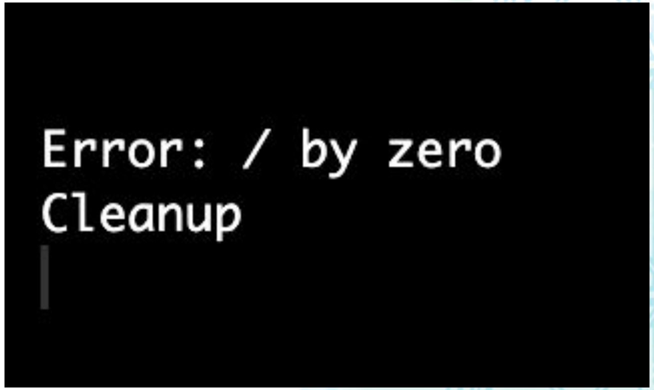
- Runtime vs compilation vs logical errors
- Handling issues with Throwable class
  - 'Error' – serious problems outside the control of the program
    - e.g. StackOverflowError
  - 'Exception' – handle errors that can be recovered from
    - e.g. IllegalArgumentException
- Errors and Exceptions are objects, not codes
  - Take advantage of and fit into OOP polymorphism

try/catch/finally

- Flow control
- try – contains risky code; catch – handles a specific type of argument
    - finally – code always executes



```java
public static void main(String args[]) {

    try {
        int result = 1 / 0;
    } catch (ArithmeticException e) {
        System.out.println("Error: " + e.getMessage());
    } finally {
        System.out.println("Cleanup");
    }

}
```



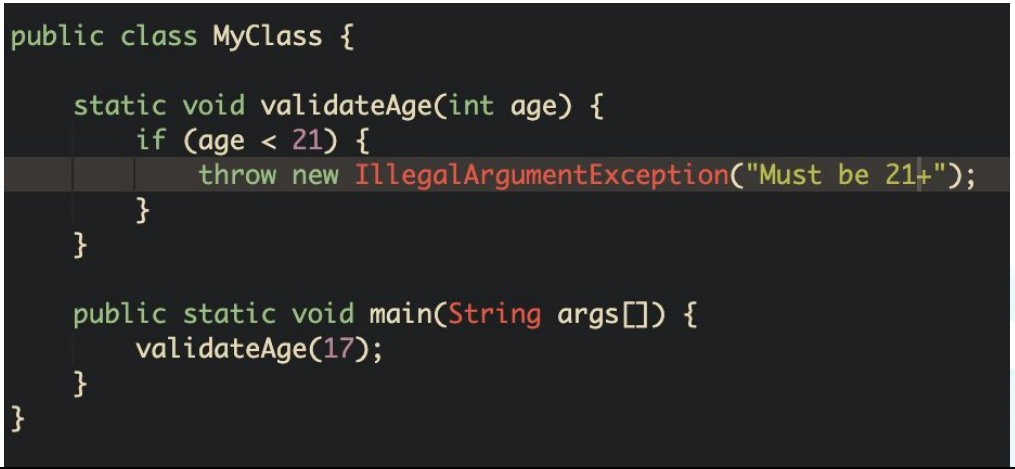
```
Error: / by zero
Cleanup
```

‘finally’ Keyword

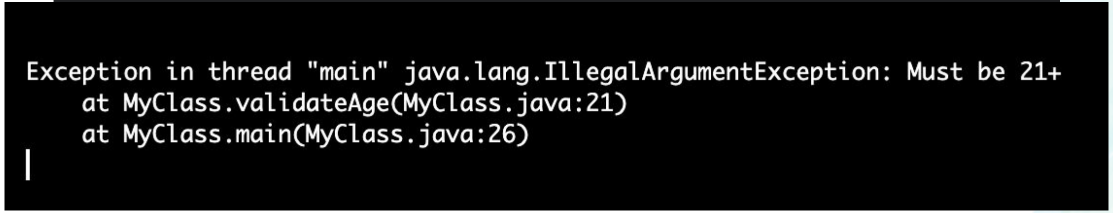
- Used for cleanup for resources
    - Resource: objects not managed by JVM
        - e.g. file, socket, database connection
- Optional if used without resources
    - e.g. the divide by zero snippet in the previous slides

throw

- Explicitly raise an exception



```java
public class MyClass {

    static void validateAge(int age) {
        if (age < 21) {
            throw new IllegalArgumentException("Must be 21+");
        }
    }

    public static void main(String args[]) {
        validateAge(17);
    }
}
```



```
Exception in thread "main" java.lang.IllegalArgumentException: Must be 21+
    at MyClass.validateAge(MyClass.java:21)
    at MyClass.main(MyClass.java:26)
```

Best Practices

- Catch the most specific exception possible
  - Exceptions **carry data**, take advantage of that!
- Use finally or try-with-resources for cleanup
- Use custom exceptions wisely
  - We can thus define application/context/domain-specific errors

Agenda

- TA introduction
- Exceptions
- **Generics**
- Collections
- Boxing/Unboxing
- File I/O

Generics

- Allows *types* to be set as parameters in classes, interfaces, and methods
- Generic classes, interfaces, methods
  - Bounded generics
- Helps compiler prevent casting and runtime errors

A Simple Example

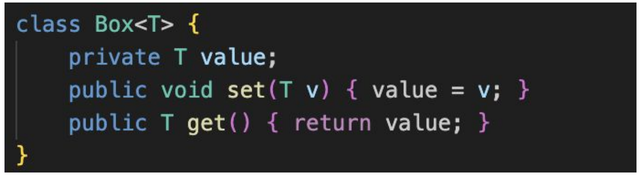
The following code snippet without generics requires casting:

```
List list = new ArrayList();
list.add("hello");
String s = (String) list.get(0);
```

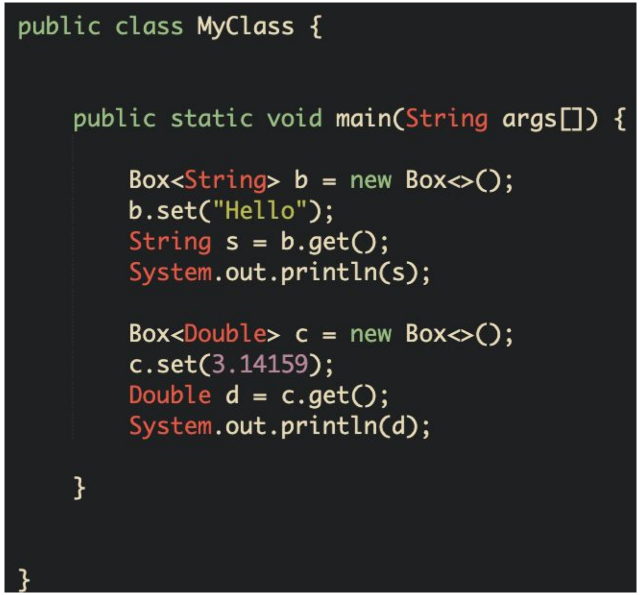
When re-written to use generics, the code does not require casting:

```
List<String> list = new ArrayList<String>();
list.add("hello");
String s = list.get(0);   // no cast
```

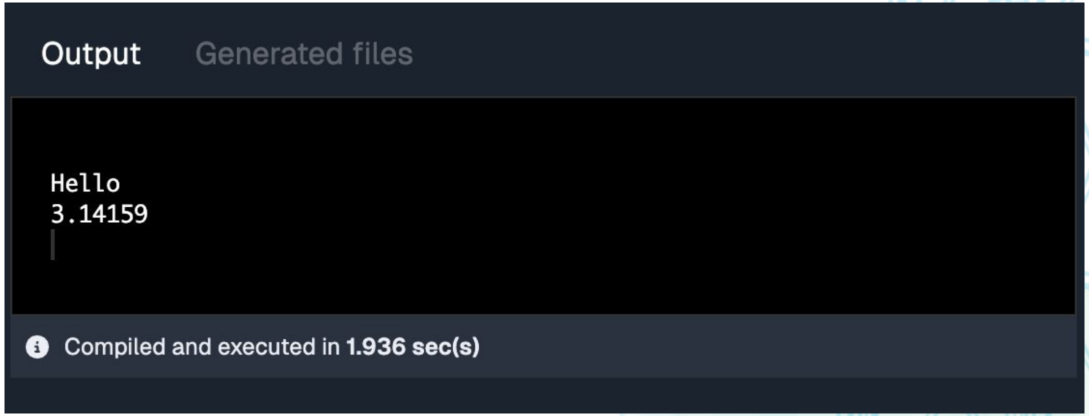
https://www.jdoodle.com/online-java-compiler



```java
class Box<T> {
    private T value;
    public void set(T v) { value = v; }
    public T get() { return value; }
}
```



```java
public class MyClass {


    public static void main(String args[]) {

        Box<String> b = new Box<>();
        b.set("Hello");
        String s = b.get();
        System.out.println(s);

        Box<Double> c = new Box<>();
        c.set(3.14159);
        Double d = c.get();
        System.out.println(d);

    }


}
```



```
Hello
3.14159
```

Compiled and executed in **1.936 sec(s)**

UCDAVIS

Agenda

- TA introduction
- Exceptions
- Generics
- **Collections**
- Boxing/Unboxing
- File I/O

Collections

- Collection interfaces
  - Lists<E>, Sets<E>, Maps<K,V>, Queues<E>/Deques<E>
    - These are different *implementations* of the interfaces, meaning they behave as and fulfill the required methods that interfaces promise to include
    - But have been implemented differently "under the hood", and may differ in aspects such as time complexity/performance
- Limitations of Java Array
  - Fixed size, cannot be dynamically sized
  - Limited built-in methods compared to collections

Lists

- ArrayList
  - Fast random operations, slow when inserting in middle
- LinkedList
  - Fast inserts/deletes, slow random access
- Methods
  - .add(), .get(), .remove(), .contains()

Sets

- HashSet
  - Hash table
- TreeSet
  - Red-black tree: guaranteed O(log n) operations
- LinkedHashSet
  - Preserves insertion order
- EnumSet

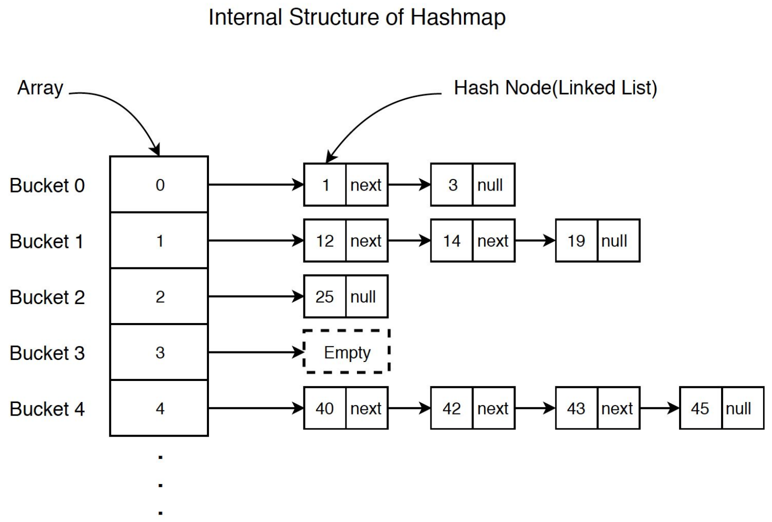
Maps

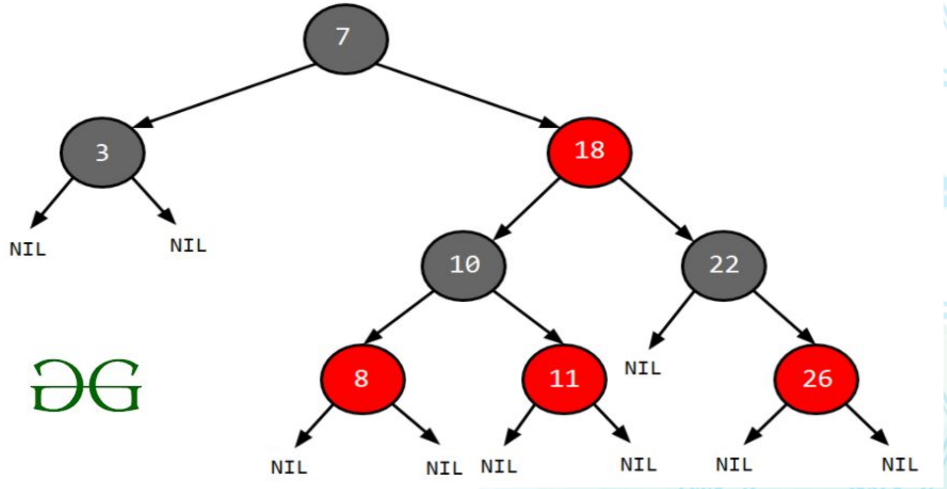
- HashMap
- TreeMap
- LinkedHashMap
- Methods
    - .put(), get(), .containsKey(), .remove()

UCDAVIS

HashMap vs TreeMap

Source: https://jojozhuang.github.io/algorithm/data-structure-hashmap/ | https://www.geeksforgeeks.org/java/internal-working-of-treemap-in-java/





Agenda

- TA introduction
- Exceptions
- Generics
- Collections
- **Boxing/Unboxing**
- File I/O

Boxing & Unboxing

- Primitives and wrappers (list on next slide)
    - Boxing: primitive -> returns wrapper
        - Integer A = 1; Double B = 2.0; Character C = ‘C’;
    - Unboxing: wrapper -> returns primitive
        - A.intValue(); B.doubleValue(); C.charValue();
- Performance costs
    - Primitives have far less overhead

Primitives and Wrappers

- Byte / byte
- Short / short
- Long / long
- Integer / int
- Long / long
- Float / float
- Double / double
- Character / char
- Boolean / boolean

Motivation

- Primitives are lightweight
  - Direct representation in stack
- Collections and generics live on the heap
  - Generics cannot support primitives, because they need references
- A wrapper is a full object
  - Contains the actual primitive
  - Also has methods, e.g. toString()
- Wrappers cost more memory

Stack vs Heap Review

- Stack
    - LIFO Memory region used for method calls, local variables
    - Primitives live here
    - Easier access
- Heap
    - Stores objects, which have dynamic memory
    - Managed with garbage collection
    - More expensive to access

Agenda

- TA introduction
- Exceptions
- Generics
- Collections
- Boxing/Unboxing
- **File I/O**

File I/O

- Persistence – files last beyond programs
- Use cases:
  - Saving data
  - Logs
  - Saving configurations

UCDAVIS

Basics

- File
  - Represents a path
- FileReader
  - Represents text
- FileInputStream, FileOutputStream
  - Represents bytes

General Procedure

- Use try
  - File handling risky
- Close resources
  - OS-level
- Try-with-resources

```
static String readFirstLineFromFileWithFinallyBlock(String path) throws IOException {

    FileReader fr = new FileReader(path);
    BufferedReader br = new BufferedReader(fr);
    try {
        return br.readLine();
    } finally {
        br.close();
        fr.close();
    }
}
```

```
static String readFirstLineFromFile(String path) throws IOException {
    try (FileReader fr = new FileReader(path);
         BufferedReader br = new BufferedReader(fr)) {
        return br.readLine();
    }
}
```

Source: https://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html

UCDAVIS