COMP10062: Week 11 Guide

Sam Scott, Mohawk College, 2022

0 Reading for this Week

For this week, you should read **sections 9.1, 9.3 (just pp. 700-702), 9.4 (just pp. 729-732) and 12.1**.

1. Catching Exceptions (Read section 9.1 and pp. 700-702)

There are three basic kinds of errors…

1. **Syntax Error:** Error that prevents compilation.
2. **Run-time Error:** Program compiles, but execution halts unexpectedly.
3. **Logic Error:** Program compiles and runs to completion, but does something unexpected or wrong due to bad logic in the code.

Exceptions are a special kind of run-time error that can be handled gracefully by the programmer.

Exceptions are objects in Java.

Exceptions are **thrown** by the method that gave rise to the error. The calling method can **throw** it onwards to *its* calling method, or it can **catch** it and handle it.

Exception Thrown

An exception causes the current method to halt immediately and "throw" the exception to the method that called it.

```
public static int getInput(int max) {
    Scanner sc = new Scanner(System.in);
    int x = 0;
    do {
        System.out.println("Enter an int no greater than " + max);
        x = sc.nextInt(); // might throw an exception
    } while (x > max);
    return x;
}
```

If the `main` method throws an exception, the program terminates and the JRE will print its **stack trace** to a special error stream (`System.err`).

```
public static void main(String[] args) {
    int x = getInput(100); // might throw an exception
    System.out.println(x);
}
```

See **Except1.java** on Canvas for the code from this example.

Catching an Exception

If you don't want the program to halt, you can catch an exception using a **`try`**…**`catch`** statement. Wrap the lines of code that might throw the exception in the `try` block, and then add a `catch` block, as shown below.

```
// some code goes here
try {
    // code that might throw an exception goes here
} catch (Exception e) {
    // code that handles the error here
}
// no matter what happens above, execution will continue here
```

If any exception happens in the `try` block above, it stops executing immediately and control jumps to the `catch` block. When the catch block finishes, the code will continue.

A `catch` block is like a method with a single parameter for an `Exception` object.

Every `Exception` object contains a stack trace to show where the exception happened and a `String` message that provides more information about what went wrong. Use `.printStackTrace()` to dump the stack trace to standard output. Use `.getMessage()` to retrieve the message.

Example: Multiple Catch Blocks

The `try` block below contains 3 lines of code. Each could throw a different type of exception. The `try` block is followed by a `catch` block for each exception type. See **Except2.java** on Canvas.

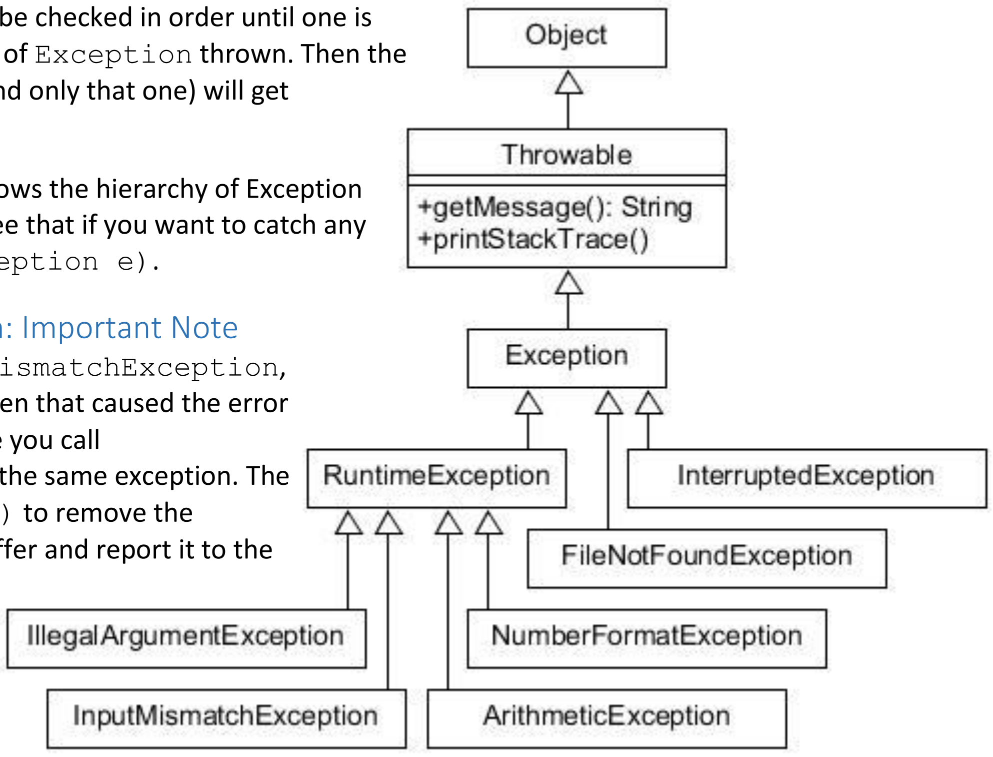
```
public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    try {
        Thread.sleep(1000);
        int x = sc.nextInt();
        System.out.println("The inverse of " + x + " is " + 1 / x);
    }

    catch (InterruptedException e) {
        System.out.println("Caught an exception!");
    }

    catch (InputMismatchException e) {
        System.out.println("Bad input! " + sc.next());
    }

    catch (ArithmeticException e) {
        System.out.println("Zero has no inverse (Error message: '"
           + e.getMessage() + "').");
    }
    System.out.println("Done. Goodbye!");
}
```

If an exception happens in the `try` block, control will jump to the `catch` blocks. They will be checked in order until one is found that matches the type of `Exception` thrown. Then the first matching catch block (and only that one) will get executed.

The UML diagram at right shows the hierarchy of Exception classes. From this, you can see that if you want to catch any exception, use `catch(Exception e)`.

InputMismatchException: Important Note

When you catch an `InputMismatchException`, the `Scanner` leaves the token that caused the error in its buffer. So the next time you call `sc.nextInt()`, you'll get the same exception. The code above calls `sc.next()` to remove the offending token from the buffer and report it to the user.



Checked and Unchecked Exceptions (pp. 700-702)

Exceptions that extend `RuntimeException` are **unchecked exceptions**.

Exceptions that extend `Exception` directly are **checked exceptions**. Examples include `InterruptedException` and `FileNotFoundException`.

If your code calls a method that might throw a checked exception, you must satisfy the **Catch or Specify Requirement** (or get a Syntax Error). This means that you must either **catch** it or **specify** in the method header that you will throw it.

To **specify**, add a `throws` declaration, as shown in the example below.

```
public static void main(String[] args) throws InterruptedException {
    Thread.sleep(1000);
}
```

2. Throwing Exceptions (Read section 9.1)

Sometimes it makes sense to create and throw an exception from a method. It's a generic way of aborting a method when something goes wrong. Throwing an exception increases the reusability of the code because it leaves it up to the application programmer to decide what to do when an error happens.

Throwing an Exception

The `Scanner` method `nextInt()` throws an `InputMismatchException` if the user types something that is not an integer. The `InputMismatchException` is an object that is created in the `nextInt()` method and then thrown with the `throw` keyword, maybe like this:

```
    InputMismatchException e = new InputMismatchException();
    throw e;
```

Or maybe like this:

```
    throw new InputMismatchException();
```

The `throw` statement is a bit like `return`. Execution halts immediately and control returns to the calllng method. If the exception is not caught, it gets thrown again from that point.

Which Exceptions Should You Throw

You can create and throw an exception of any type (or you can create your own exception types as we shall see) but the one type you will probably use most is `IllegalArgumentException`.

Whenever a user calls a method of yours and there is something wrong with one of the arguments passed to your parameters, you can throw one of these.

```
    if (x < 0)
        throw new IllegalArgumentException();
```

It's always a good idea to provide a message as well, which you can do by using a different constructor:

```
    if (x < 0)
        throw new IllegalArgumentException("X must be positive.");
```

If this exception object is caught, you can access the message it was created with using its `getMessage()` method.

3. Lists in Java (Read Section 12.1)

Arrays are useful for storing sequences of primitive values and objects, but you have to specify a fixed size in advance and it is difficult to insert and remove values.

`Lists` are like arrays, but they grow and shrink as necessary and have insertion and removal methods already written. The price is that, depending on the implementation, `Lists` can be a little slower than arrays and use a little more storage than arrays. But for most applications you won't notice much difference.

The most common type of `List` in Java is the `ArrayList`, which gets its name from the fact that under the hood, the list of items is stored in an array.

Declaring an `ArrayList`

`ArrayLists` are "generic" classes. Generic classes are associated with other classes, but you get to specify which classes they are associated with when you instantiate them. You do this by including the names of the associated classes inside angle brackets (`"<"` and `">")`.

Generically, we use `E` to stand for any possible class. So the Array List type in Java is usually written `ArrayList<E>`.

Here's a declaration for an `ArrayList<E>` variable, specifying that `String` objects will be stored. In other words, type `E` is `String`. You will need to import `java.util.ArrayList` for this to work.

```
ArrayList<String> a;              ← a is null at this point
```

Creating ArrayLists

```
a = new ArrayList<>(); ← This creates an empty ArrayList<String> object
System.out.println( a.size() );  ← will print 0 to the console (size = length)
```

Initializing ArrayLists

Unlike arrays, you don't have to set up an initial value for every element in the list because initially, there are no elements in the list.

Adding and Inserting elements

```
a.add("pascal");       ← adds an element to the end of the list
a.add(1, "basic");     ← inserts an element as index 1. Can throw an exception
```

Removing elements

```
a.remove(1);           ← removes the element at index 1
```

Accessing Objects in an ArrayList

```
a.get(2);               ← Just like a[2] for an array. Can throw an exception.
```

Processing ArrayLists

```
for (int i=0; i<a.size(); i++)    ← use an index
   System.out.println(a.get(i));

for (String e: a)                 ← use an enhanced for loop
  System.out.println(e);
```

ArrayLists, Wrappers and Autoboxing

`ArrayList<E>` objects have to store object types. `E` cannot be a primitive type like `int` or `double`. But every primitive type comes with a **wrapper** class you can use instead. They're called wrapper classes because their objects store single primitive values. The primitive is the value, the object is the wrapping.

```
ArrayList<int> b = new ArrayList<>();          ← this doesn't work!
ArrayList<Integer> b = new ArrayList<>();      ← this is ok!
```

Here's the list of all eight wrapper classes.

Primitives:	byte	short	int	long	float	double	char	boolean
Wrappers:	Byte	Short	Integer	Long	Float	Double	Character	Boolean

(You've used these classes before: `Integer.parseInt()`, `Double.parseDouble()`, etc.)

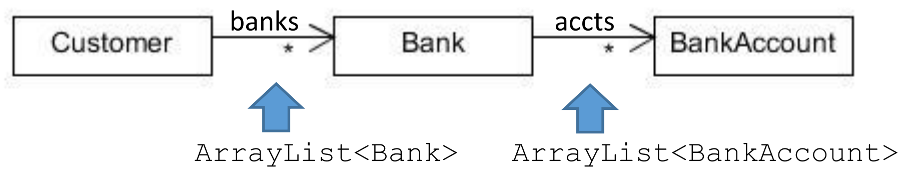
Because of a Java process called **autoboxing** you can act as if the wrapper class is just the same as the primitive type. Basically, if the compiler sees 5.6, it "boxes it up" into an instance of the `Double` class.:

```
ArrayList<Integer> b = new ArrayList<>();

b.add(3);
b.add(-5);
b.add(33);
b.add(10);

sum = 0;
for (Integer e : b)
      sum += e;
System.out.println(sum);
```

ArrayLists for Multiple Association

`ArrayList<E>` objects can be used as instance variables, just like Arrays can.

Finally, you can implement the Kleene star (`*`) multiplicity. The `*` means "0 or more", which is exactly how many objects an `ArrayList` can contain!



4. Mouse Events (Read pp. 729-732)

A mouse event is triggered every time one of the following things happens to a GUI component:

• A mouse button is pressed	(`MouseEvent.MOUSE_PRESSED`)
• A mouse button is released	(`MouseEvent.MOUSE_RELEASED`)
• The mouse moves	(`MouseEvent.MOUSE_MOVED`)
• The mouse is dragged	(`MouseEvent.MOUSE_DRAGGED`)
• The mouse enters the component	(`MouseEvent.MOUSE_ENTERED`)
• The mouse leaves the component	(`MouseEvent.MOUSE_EXITED`)
• Etc.	

Usually, you don't need to handle mouse events yourself. But when you're making a program with a `Canvas` (e.g. a game, a drawing app, an interactive graphic, etc.) it might be very nice to let the user interact directly with the drawings on the canvas instead of having to fill in text fields and press buttons.

To add a mouse event handler for a button press to a canvas named `c`, include the following line in the start method…

```
c.addEventHandler(MouseEvent.MOUSE_PRESSED, this::pressHandler);
```

- The first parameter specifies the event type using static constants defined in `MouseEvent`.
- The second parameter specifies the event handler method that should be called (just like the `Button` method `setOnAction()`).

… then create the `pressHandler` method. This must be `void`, must accept a `MouseEvent` parameter, and should probably be private.

```
private void pressHandler(MouseEvent me) { ... }
```

Unlike `ActionEvent` objects for button event handlers, which can often be ignored, you will almost certainly want to get some information from the `MouseEvent` object. At a minimum it can tell you where the mouse was when the event happened, and what button was involved.

Here's an example event handler that goes with the code above.

```
private void pressHandler(MouseEvent me) {
    System.out.println("Pressed " + me.getButton() + " at (" +
                       me.getX() + "," + me.getY() + ").");
}
```

Get **MouseEventDemo.java** from Canvas for an example.