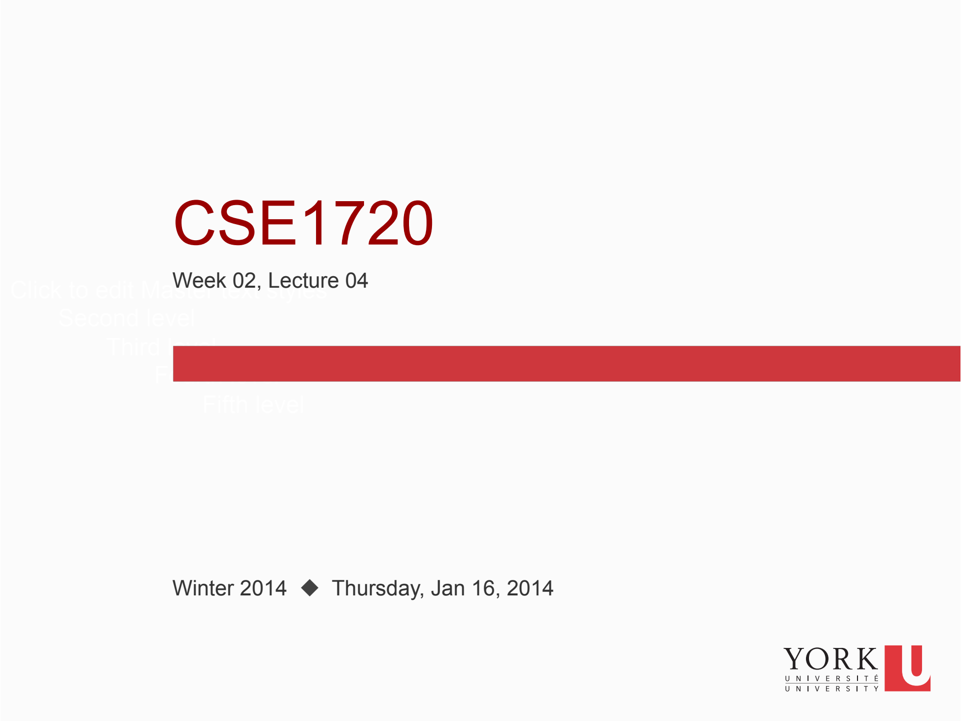
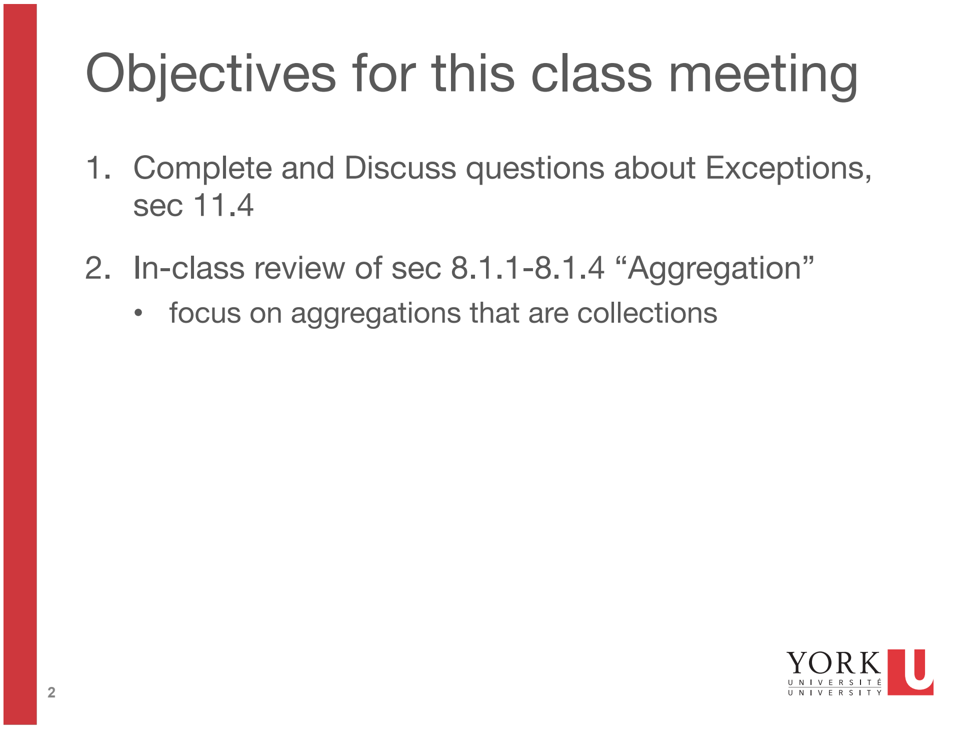


CSE1720

Week 02, Lecture 04

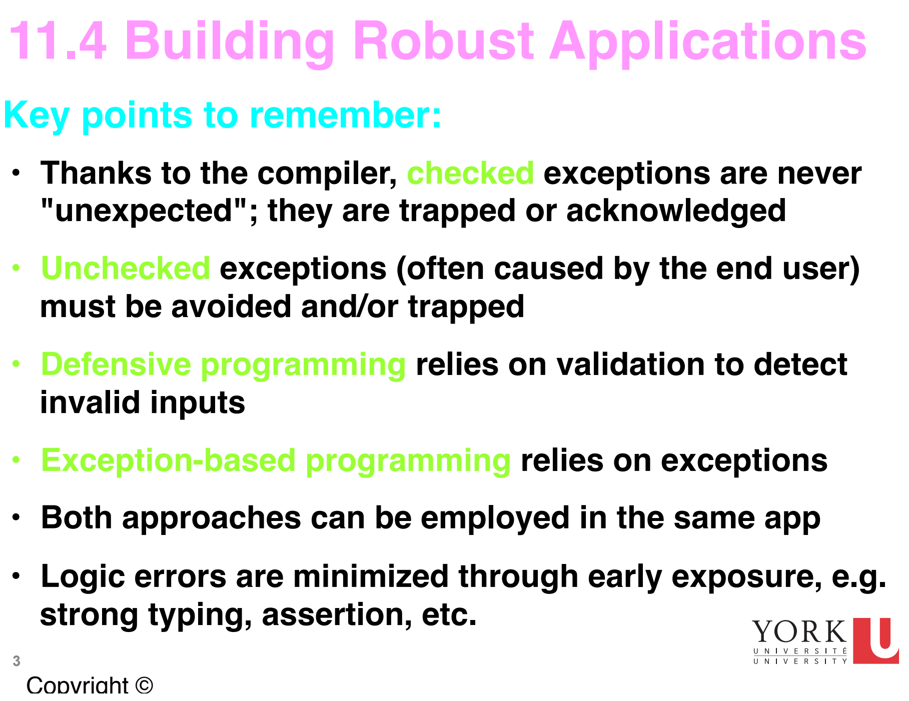
Winter 2014 ◆ Thursday, Jan 16, 2014



Objectives for this class meeting

1. Complete and Discuss questions about Exceptions, sec 11.4
2. In-class review of sec 8.1.1-8.1.4 “Aggregation”
   - focus on aggregations that are collections

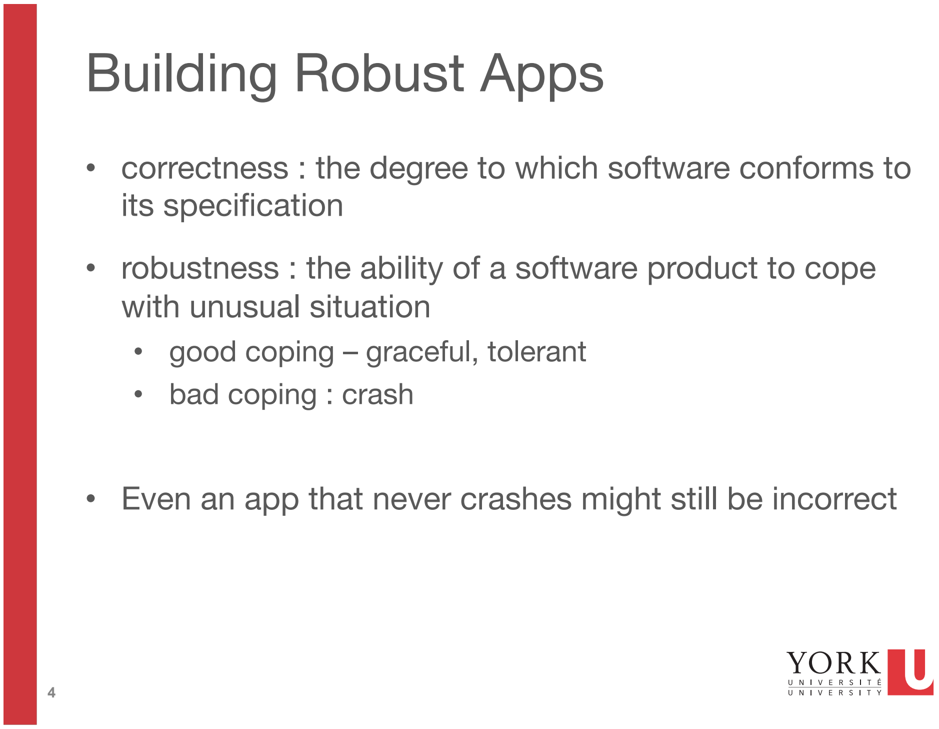
2



11.4 Building Robust Applications

Key points to remember:

- Thanks to the compiler, checked exceptions are never "unexpected"; they are trapped or acknowledged
- Unchecked exceptions (often caused by the end user) must be avoided and/or trapped
- Defensive programming relies on validation to detect invalid inputs
- Exception-based programming relies on exceptions
- Both approaches can be employed in the same app
- Logic errors are minimized through early exposure, e.g. strong typing, assertion, etc.



Building Robust Apps

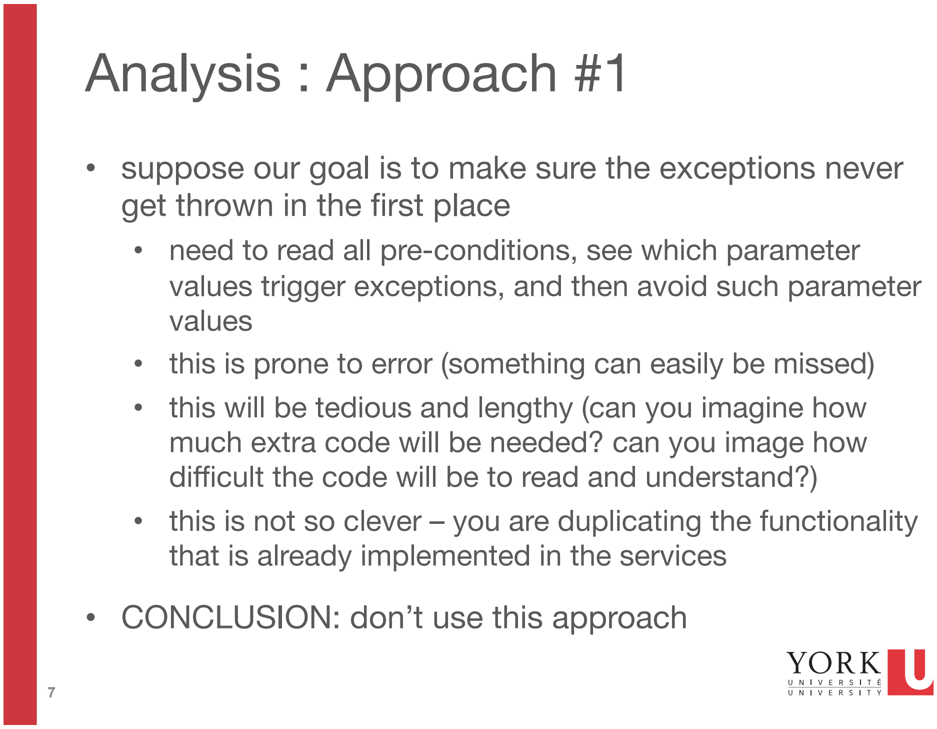
- correctness : the degree to which software conforms to its specification
- robustness : the ability of a software product to cope with unusual situation
  - good coping – graceful, tolerant
  - bad coping : crash
- Even an app that never crashes might still be incorrect

Building Robust Apps

- The goal of robustness means that we **don't want our software to crash**
- We will use all sorts of services, many of which potentially throw exceptions
- unhandled exceptions cause apps to crash
- crashing app == not **robust app.**
- Do we rely on our human abilities to track all of these potential sources of exceptions?
  - Humans make mistakes, even with the best of intentions.

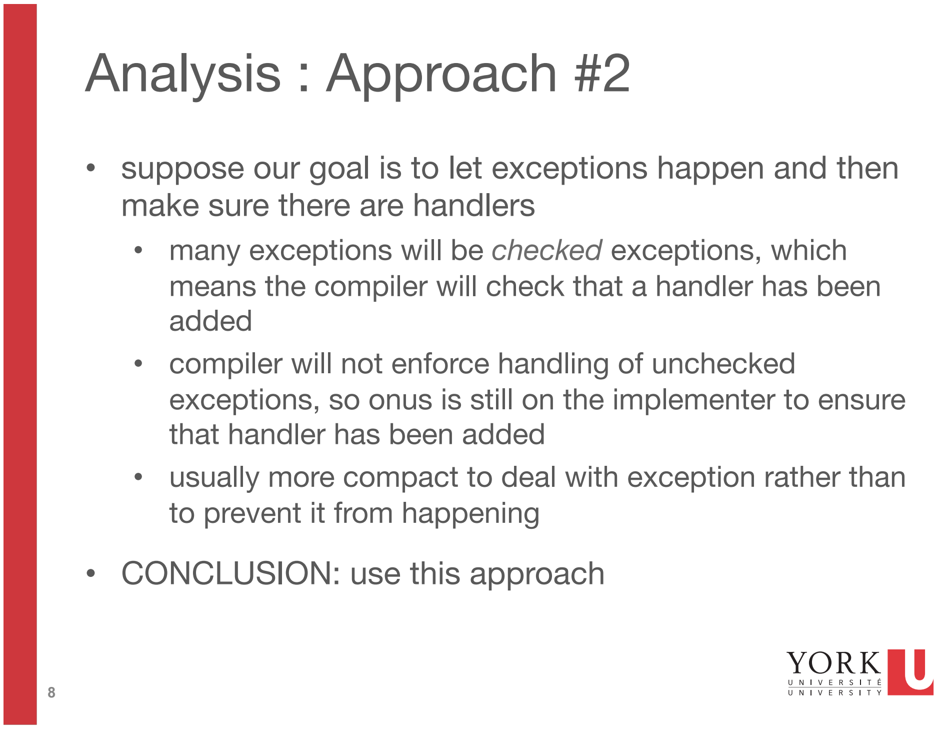
Building Robust Apps

- what approach should we use to ensure that our app doesn't crash?
- **approach #1 –** make sure the exceptions never get thrown in the first place!
  - need to read all pre-conditions, see which parameter values trigger exceptions, and then avoid such parameter values
  - build in a whole bunch of if-then clauses and other ways of validation for parameters, **before** invoking services
- **approach #2 –** let exceptions happen
  - make sure all of the necessary handlers are in place



Analysis : Approach #1

- suppose our goal is to make sure the exceptions never get thrown in the first place
  - need to read all pre-conditions, see which parameter values trigger exceptions, and then avoid such parameter values
  - this is prone to error (something can easily be missed)
  - this will be tedious and lengthy (can you imagine how much extra code will be needed? can you image how difficult the code will be to read and understand?)
  - this is not so clever – you are duplicating the functionality that is already implemented in the services
- CONCLUSION: don't use this approach



Analysis : Approach #2

- suppose our goal is to let exceptions happen and then make sure there are handlers
  - many exceptions will be *checked* exceptions, which means the compiler will check that a handler has been added
  - compiler will not enforce handling of unchecked exceptions, so onus is still on the implementer to ensure that handler has been added
  - usually more compact to deal with exception rather than to prevent it from happening
- CONCLUSION: use this approach

- topic shift into collections now

Questions about Collections

- What is a collection?
  What is an aggregate with variable multiplicity?
  How are these questions related?
- RQ8.19 What does variable multiplicity mean?
  How is aggregation depicted in UML, both with fixed and with variable multiplicity?
- RQ8.20 If a collection is statically allocated, then what should be passed to its constructor?
- RQ8.21 Can you add an element to a collection even if it is already in it?

Questions about Collections

- RQ8.22 What happens if you attempt to add an element to a full, statically-allocated collection?
- RQ8.23 What is a traversal?
- RQ8.24 How do you determine the number of elements in a collection if it supports indexed traversals?
- RQ8.25 How do you determine the number of elements in a collection if it supports iterator-based traversals?
- RQ8.26 (a) Explain how a traversal can be used to perform a search. (b) Why are traversal-based searches called exhaustive?

OK – those are many questions.

Let’s talk about some answers

The first question… What is a collection?

About Collections…

The course material concerns several topics about collections
e.g., collection traversals, static/dynamic allocation, etc.

These concepts will make a lot **more sense** if you have a crystal clear understanding about what a collection actually is

So what **is** a collection anyway?

Let's start with:

- It is a class instance (an object)
- The class instance has attributes (elements)
- The elements are non-primitive, non-String

- **These three things define an aggregation**
- **So a collection is an aggregation**
- **BUT NOT ALL AGGREGATIONS ARE COLLECTIONS**

So what **is** a collection anyway?

Let's start with what a collection is **NOT**.

A collection is **NOT** a set.

- A set is, by definition, a **collection** that does not contain duplicate elements.

A collection is **NOT** a list.

- A list is, by definition, an ordered **collection**.

You can't use the term you are trying to define in the definition!

So what **is** a collection anyway?

Instead of trying to articulate what a collection **IS** it is better to articulate what a collection **DOES**

This is a Forrest Gump way of defining something:

A collection *is* what a collection *does*

So what does a collection **do**?

1. It exists as a class instance.
2. It has elements.
   - and these elements are understood to be non-primitive, non-String
3. It allow clients to query its size
4. It allow clients add and remove elements
5. It allow clients to traverse the elements
   - at least one way must be provided, although there are several possible ways

A diagnostic test:
Is this object a collection?

Is it a class instance?

Does it have elements?

Can I traverse those elements?

Does it let me add elements?

Does it let me remove elements?

Does it tell me its size?

Then it is a collection.*

*a collection does a few other things, but we will talk about these later

Another (equivalent but different) way of defining a collection [textbook]

A collection is an **aggregate** in which the **multiplicity** is variable and in which the aggregated parts are called **elements**.

Is an array a collection?

No, according to the textbook.

An array is not an aggregate since it is not a class instance.

An **object** is a class instance or an array

Can a utility class encapsulate a collection?

No, a utility class is not a class instance.

We could *emulate* a collection

- static attributes would hold the elements
- the required operations would be provided by static methods
  - access the size of the collection (number of elements)
  - addition and removal of elements
  - traversal of elements

Some examples

- Suppose our elements are the colours of the rainbow
- We will use the class `java.awt.Color` to encapsulate each colour

```
Color red = new Color(255, 0, 0);
Color orange = new Color(255, 165, 0);
Color yellow = new Color(255, 255, 0);
Color green = new Color(0, 255, 0);
Color blue = new Color(0, 0, 255);
Color purple = new Color(128, 0, 128);
```

Using an array…

- Refer to code example `L04Ex01`

```
Color[] theRainbow = new Color[6];
```

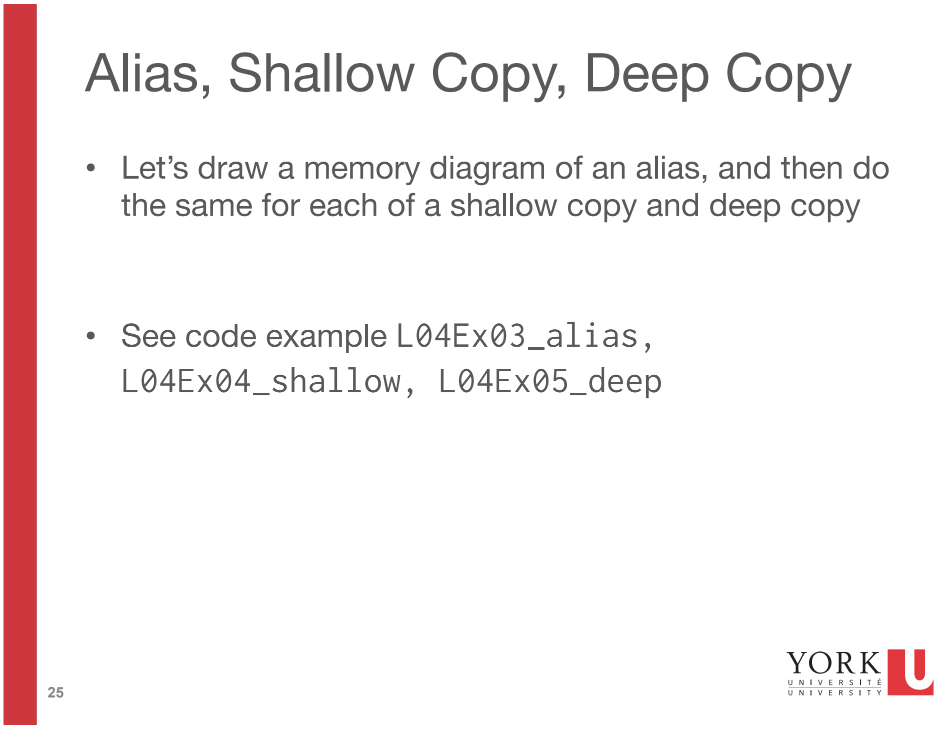
- can I add more elements to this array object?

Using an collection…

- Refer to code example `L04Ex02`

```
ArrayList<Color> theRainbow1 = new ArrayList<Color>();
```

- can I add more elements to an `ArrayList` collection?



Alias, Shallow Copy, Deep Copy

- Let’s draw a memory diagram of an alias, and then do the same for each of a shallow copy and deep copy
- See code example `L04Ex03_alias`, `L04Ex04_shallow`, `L04Ex05_deep`