2IPC0 Programming Methods

From Small to Large Programs

Loek Cleophas
Eindhoven University of Technology
Department of Mathematics & Computer Science
Software Engineering & Technology Group

http://canvas.tue.nl/courses/473

Recap Lecture 3

- The contract of a robust method specifies behavior in *all* cases. When precondition is violated, an `Exception` is thrown.

- Exceptions provide a mechanism to bypass normal control flow, in case of *failures* or *special situations*, to *inform* and *avoid harm*.

- Java exceptions involve:

  – objects that are instances of `Exception` or its subclasses

  – **`throws`** clauses in method headers

  – contracts that specify 'which exceptions are thrown when', by `@throws` tags in javadoc comments

  – **`throw`** statements

  – **`try`** ... **`catch`** ... **`finally`** statements

Overview

- Exceptions in Java*
- Runtime Assertion Checking (RAC)
- Design patterns and other forms of reuse
- Singleton Pattern, global variables

*Reused and adapted some slide material from Alexandre Denault

Robustness

?

Robustness

- Behave predictably under all circumstances
- Check preconditions, and signal violations explicitly

  Use exceptions or Runtime Assertion Checking (RAC)

Defining Exceptions

- The following is minimal (does not allow a message):

```java
public class MyException extends Exception { }
```

- To have a constructor with a message, you need:

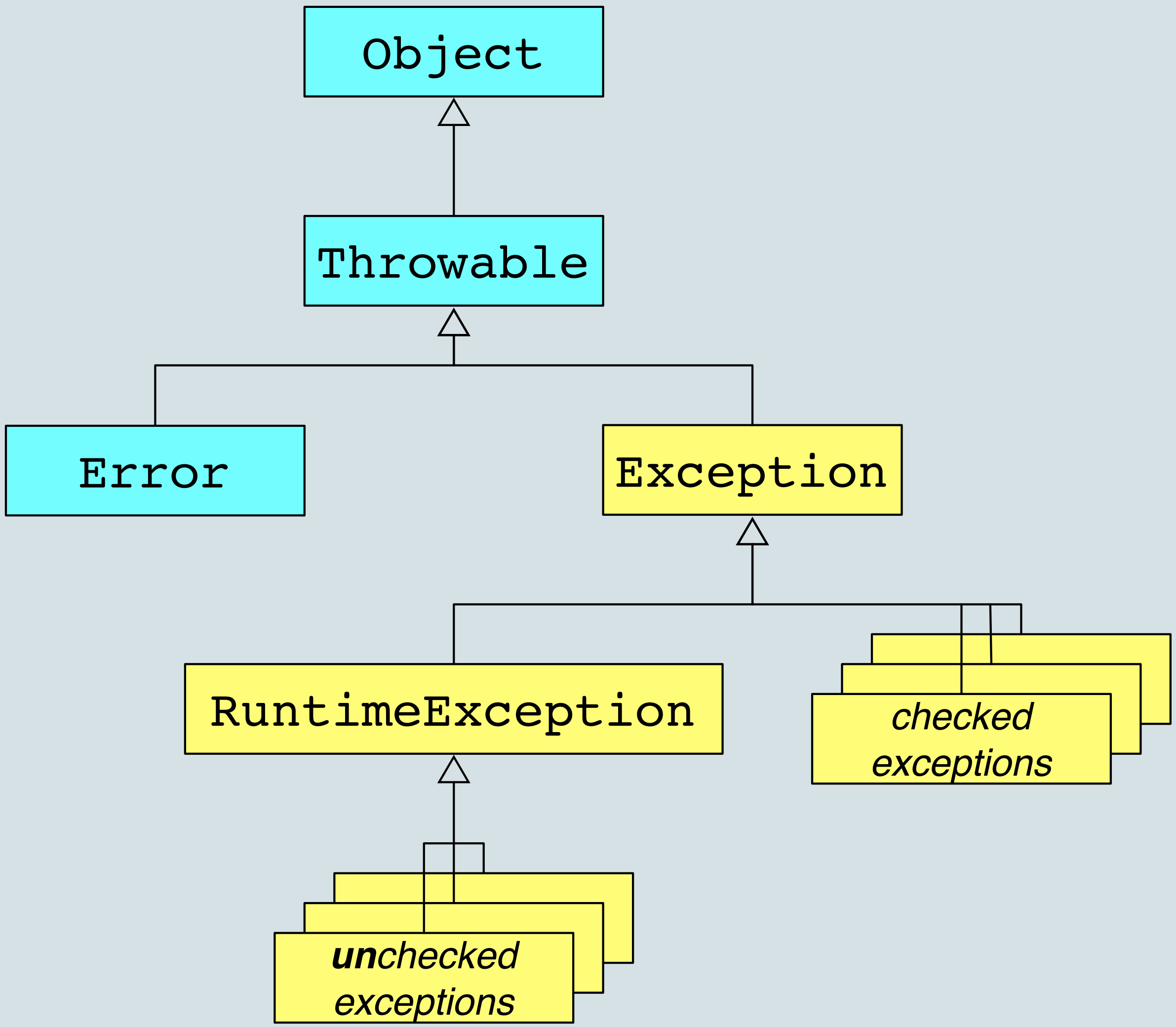
```java
public class MyException extends Exception {
    MyException() { super(); }
    MyException(String s) { super(s); }
}
```

- Exceptions may be much more elaborate:

```java
public class MyException extends Exception {
  Object offensiveObject;
  MyException(String s, Object o) { super(s); offensiveObject = o; }
  Object getOffensiveObject() { return offensiveObject; }
}
```

Exception Type Hierarchy in Package `java.lang`

In Java,
Exceptions
are objects,
existing in the
object hierarchy:



Defining Exceptions (2)

- You can use `Exception` and `RuntimeException` directly, without using or defining a (new) subtype exception.

  But this is not good programming style, because it does not convey much information (is too vague, too general).

- You can use other predefined exceptions, when appropriate.
  E.g. `NullPointerException`, `IndexOutOfBoundsException`

- `...Exception` naming is not enforced, but this is good practice.

- In 2IPC0, use `IllegalArgumentException` or `IllegalStateException` to signal precondition violations.

Where to Define Exception Types?

- Some Exceptions occur in many packages (e.g.: `NotFoundException`).
- It makes sense to avoid *naming conflicts* and define a separate Exception package.
- However, if special Exceptions are thrown by your package, you can define them inside your package.

Kinds of Exceptions: Checked

Checked exceptions: subclass of `Exception` but not of `RuntimeException`

- *must be listed* in the **`throws`** clause of the method that can throw the exception.
- *must be handled* by the caller code, either by
  - propagation, or
  - catching,

  otherwise, a compile-time error occurs.
- *typically, used to signal recoverable special situations*.

Checked Exceptions: Example

`java.util.Scanner(...)` can throw `FileNotFoundException`

```java
1 import java.io.File;
2 import java.util.Scanner;
3
4 public class CheckedExceptions {
5
6     public void doesNotCompile() {
7         Scanner scanner = new Scanner(new File("file.txt"));
8     }
9
10 }
```

```
Error: .../CheckedExceptions.java:7: unreported exception
java.io.FileNotFoundException;
must be caught or declared to be thrown
```

Checked Exceptions: Example (cont'd)

```java
import java.io.File;
import java.util.Scanner;

public class CheckedExceptions {

    public void doesCompile() throws Exception {
        Scanner scanner = new Scanner(new File("file.txt"));
    }

}
```

Client code that calls `doesCompile` is forced to 'handle' the exception.

Kinds of Exceptions: Unchecked

Unchecked exceptions: `RuntimeException` and subclasses

- *don't have to be listed* in the **`throws`** clause.

  But it is good practice to list unchecked exceptions that can be thrown, especially if it can be expected that the caller can usefully catch the exception.

- *don't have to be handled* explicitly by the caller.

- *typically, used to signal non-recoverable failures.*

Examples: `NullPointerException`, `IndexOutOfBoundsException`

Handling Exceptions Implicitly

- If method `m` calls method `p` without wrapping the call in a **`try`** block, then an exception thrown by `p` aborts execution of `m` and the exception is propagated to the method that called `m`.
  (A calls B calls C calls D throws e, A catches e)
- To propagate a *checked* exception, the calling method must list the thrown exception. (If not, a compile error results.)
- *Unchecked* exceptions are automatically propagated until they reach an appropriate **`catch`** block. (Not enforced by compiler)
- But you can still list unchecked exceptions in the header, and it is good practice to do so.
  (The client is made aware of the unchecked exception and can catch it if desired.)

Programming with Exceptions

- How to handle thrown exceptions?

- Possibilities:

  - Handle specifically: separate **`catch`** blocks deal with each situation in a different way
  - Handle generically: one **`catch`** block for supertype exception takes generic action like println and halt or restart program from earlier state
  - Reflect the exception: the caller also terminates by throwing an exception, either implicitly by propagation or explicitly by throwing a different exception (usually better)
  - Mask the exception: the caller catches the exception, ignores it, and continues with normal flow

Reflection (a.k.a. Translation)

```java
// Coding Standard violated to fit on one slide
    /** @throws NullPointerException  if a == null
      * @throws EmptyException  if a.length == 0
      * @pre a != null && a.length != 0
      * @post \result == (\min i; a.has(i); a[i]) */
    public static int min (int [ ] a)
            throws NullPointerException, EmptyException {
        int m; // minimum of elements of a
        try {
            m = a[0];
        } catch (IndexOutOfBoundsException e) {
            throw new EmptyException("Arrays.min.pre violated");
        }
        for (int element : a) {
            if (element < m) { m = element; }
        }
        return m;
    }
```

Masking

```
// Coding Standard violated to fit on one slide
    /** @throws NullPointerException  if a == null
      * @pre a != null
      * @post \result == a is sorted in ascending order */
    public static boolean sorted (int [ ] a)
            throws NullPointerException {

        int prev;

        try { prev = a[0]; }
        catch (IndexOutOfBoundsException e) { return true; }

        for (int element : a) {
            if (prev <= element) { prev = element; }
            else { return false; }
        }
        return true;
     }
```

Design Issues

- When to throw exceptions?
  - To make method contracts *robust*.
  - To avoid encoding information in ordinary or extra results.
  - To signal special —usually erroneous— situations, often in non-local use of functions (that is, propagating across calls).
  - To guarantee that detected failures cannot be ignored.
- When *not* to throw exceptions?
  - When the context of use is local (consider use of `assert`).
  - When the precondition is too expensive or impossible to check.

Defensive Programming

- *Defensive programming* is the attitude to include run-time checks for bad situations that should never occur (but still could occur due to “unforeseen” circumstances: environment, other programmers). These situations are usually not covered in specifications.

- Exceptions or `assert` statements can be used, because they do not burden the normal control flow.

- Typically, you could use one generic unchecked exception.
  E.g. `FailureException`

```
if (unimaginable) {
    throw new FailureException(
        "class C, method M: the unimaginable happened");
}
```

Exceptions versus `assert` statements

Runtime Assertion Checking (RAC)

- `assert` *booleanExpr*
- `assert` *booleanExpr* : *stringExpr*
- **`if`** `( !` *booleanExpr* `)` **`throw new`** `AssertionError(`*stringExpr*`);`
- Note that `AssertionError` is an `Error`, not an `Exception`.
  Cannot be caught; always aborts execution; cannot unit test it.
- Execution of `assert` is disabled by default; to enable:
  - in DrJava: Preferences > Miscellaneous > JVMs:
    JVM Args for Main JVM $\boxed{\text{-ea}}$
  - in NetBeans: Project Properties > Run > VM Options: $\boxed{\text{-ea}}$

When to Throw Exceptions in This Course?

- *Non-private* methods:
  - Strive for robustness
    Except when performance would suffer disproportionately.
  - Contracts must explicitly state conditions for exceptions.
    Use **`if`** and throw an exception, e.g. `IllegalArgumentException`
  - Can use `assert` for situations not covered by exceptions.
- *Private* methods:
  - May have stronger precondition without explicit exceptions.
  - You are encouraged to use `assert` to check precondition.

Checked versus Unchecked Exceptions

- Advantages of checked exceptions:
  - Compiler enforces proper use (code must catch or propagate).
  - Prevents wrongly captured exceptions.
- Disadvantages of checked exceptions:
  - Forces developer to deal with them explicitly, even in cases where they provably will not occur.
- Use unchecked exceptions if you expect they will not occur, because they can be conveniently and inexpensively avoided.
- Otherwise, use checked exceptions; i.e., when they cannot be avoided easily.

How to Handle Exceptions in This Course?

- In unit tests: Always check that exceptions are thrown properly according to the contract.
- In non-test code: Never catch unchecked exceptions, *unless* ...
- ... there is an important reason and good way for recovering.

  Unchecked exceptions should signal the presence of a defect.
- Checked exceptions must be handled: recover or propagate.

  Checked exceptions should signal a recoverable special situation, typically from using a standard library method (e.g. for I/O).

Design Patterns

- A Design Pattern is an *outline* of a *general* solution to a design problem, *reusable* in *multiple, diverse, specific* contexts.

- Distinction between forms of reuse (cf. Ch.1 of Burris, and DRY):

  - *coding idiom*
  - *code library*
  - *code framework*

    (Hollywood principle: ‘don’t call us, we call you’)
  - *design pattern*
  - *architectural pattern*

Taxonomy of Design Patterns

- **Creational** patterns

  Singleton, Factory Method

- **Structural** patterns

  Adapter, Decorator, Composite, Façade

- **Behavioral** patterns

  Strategy, Template Method, Iterator, Observer, Command, State

- **Concurrency** patterns

  "`SwingWorker`"

Singleton Pattern Motivation

- One Controller
- One Undo-Redo facility
- One Logger

Existence of multiple Controllers, . . . , could be problematic.

Concern: How to prevent creation of multiple instances?

Singleton Anti-Pattern: Global Variable

Ignore the issue; rely on client code to create only one instance, in a global variable:

```
/** The one-and-only global logger object. */
public static final Logger theLogger = new Logger();
    // Better: make private and provide accessor
```

used as

```
Main.theLogger.log(name + " received " + arg);
```

See `SingletonExamples.zip`: `SingletonAntiPatternGlobalVariable`

Singleton Anti-Pattern: Class with Statics

Make all instance variables and methods in the class **`static`**, and use the class itself, not an object.

Optionally, make class **`abstract`**, to prohibit instantiation

Con: Cannot use as parameter; cannot override its methods

See `SingletonExamples.zip`: `SingletonAntiPatternStatic`

Intent

The Singleton design pattern

- ensures that *not more than one instance* of a class is created;
- it provides a *global point of access* to this instance.

Singleton Pattern Solution (adapted from Burris)

- Make the constructor of the class **`private`**

  to prevent clients from creating instances [...] directly.

- Add a **`public static`** method `getInstance()`

  that returns an instance of the class.

- The first time `getInstance()` is called

  an instance of the class is created, cached, and returned.

- On subsequent calls, the cached instance is returned.

Singleton Design Pattern Example: Singleton Logger

```
public class Logger {

    /** The one-and-only global logger. */
    private static Logger theLogger;

    /** The private contructor. */
    private Logger() { }

    /** Returns the one-and-only global logger. */
    public static Logger getLogger() {
        if (theLogger == null) {
            theLogger = new Logger();
        }
        return theLogger;
    }
     ... other logger stuff ...
}
```

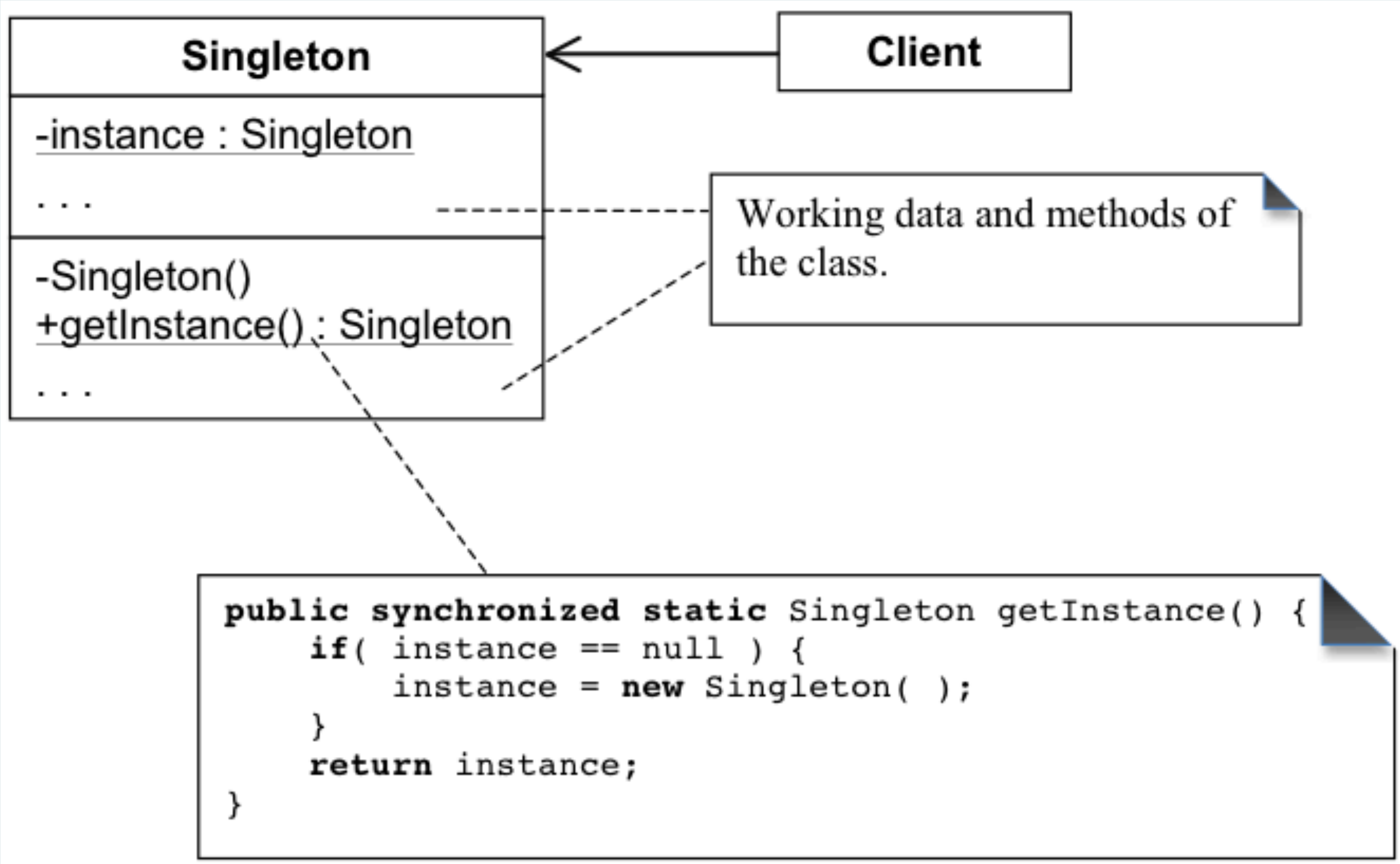
See `SingletonExamples.zip`: `SingletonLogger`

Singleton Design Pattern Example: Client Code

The singleton logger is used as:

```
Logger.getLogger().log(name + " received " + arg);
```

Singleton Pattern Class Diagram (from Burris)



Singleton Pattern and Concurrency

Concurrency in multithreaded applications, without **synchronized**

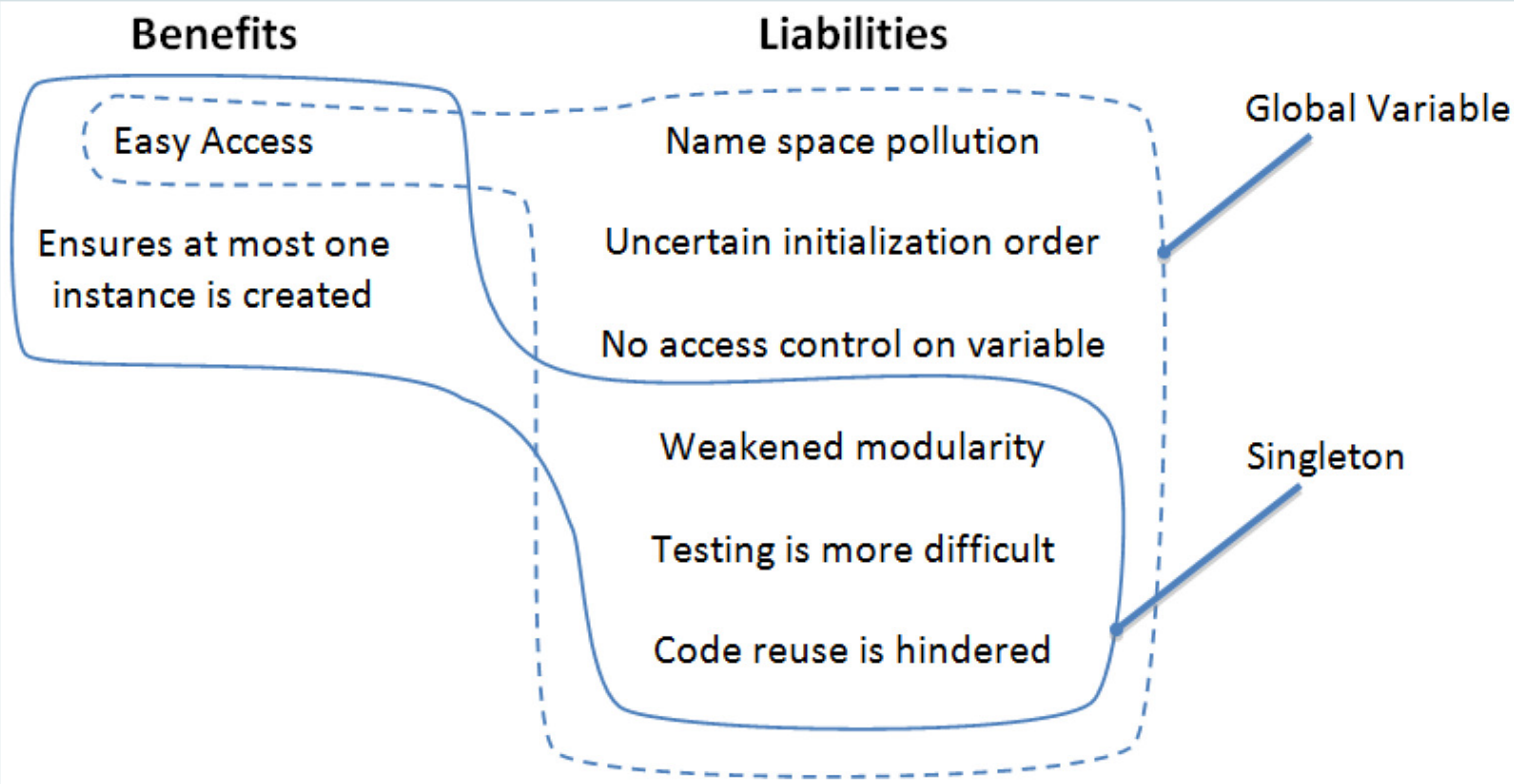
```
    private static Singleton instance;

    /** This factory method is not thread-safe! */
    public static Singleton getInstance() {
        if ( instance == null ) {
            instance = new Singleton();
        }
        return instance;
    }
```

Do you see the problem (a so-called race condition)?

synchronized ensures mutual exclusion, but adds runtime overhead

Singleton Pattern Benefits & Liabilities (from Burris)



Singleton Pattern Benefits & Liabilities (explained)

Weakened modularity Many other modules will depend on the global object/variable

Testing is more difficult Because tests must also work with that one global/object; cannot easily vary it

Code reuse is hindered To reuse code depending on the global, you must also reuse that global/object

How to Avoid Dependency on Globals

- Modules can depend directly on global constants, variables, and singleton objects: hinders testing and reuse (cannot vary them).

- To avoid dependence on external global constants and variables, you can *parameterize* modules, and supply actual concrete globals as argument.

- To avoid dependence on global objects, apply Strategy pattern, and invert dependencies.

  Make other modules depend on an (abstract) interface; hence, they do not depend on the concrete globals.

  Provide actual object at runtime.

  See `SingletonExamples.zip`: `StrategyPatternToInvertDependence`

Assignments Series 2

- Refresher: consult book by Eck: 3.7, 8.3, 8.4.1
- From Design Patterns book by Burris: Ch.1, 2, 7
- Apply Test-Driven Development, including Exceptions, to `CountDigitsWithRadix` and `Powerize`

Summary

- Java exceptions:
  - Checked versus Unchecked (`RuntimeException`)
  - Handle specifically/generally, Reflect (Translate), Mask
- `assert` throws an `Error`; terminates execution, cannot be caught.

  Runtime Assertion Checking (RAC)
- Design patterns and other forms of reuse
- Singleton Pattern, global variables

  See `SingletonExamples.zip`