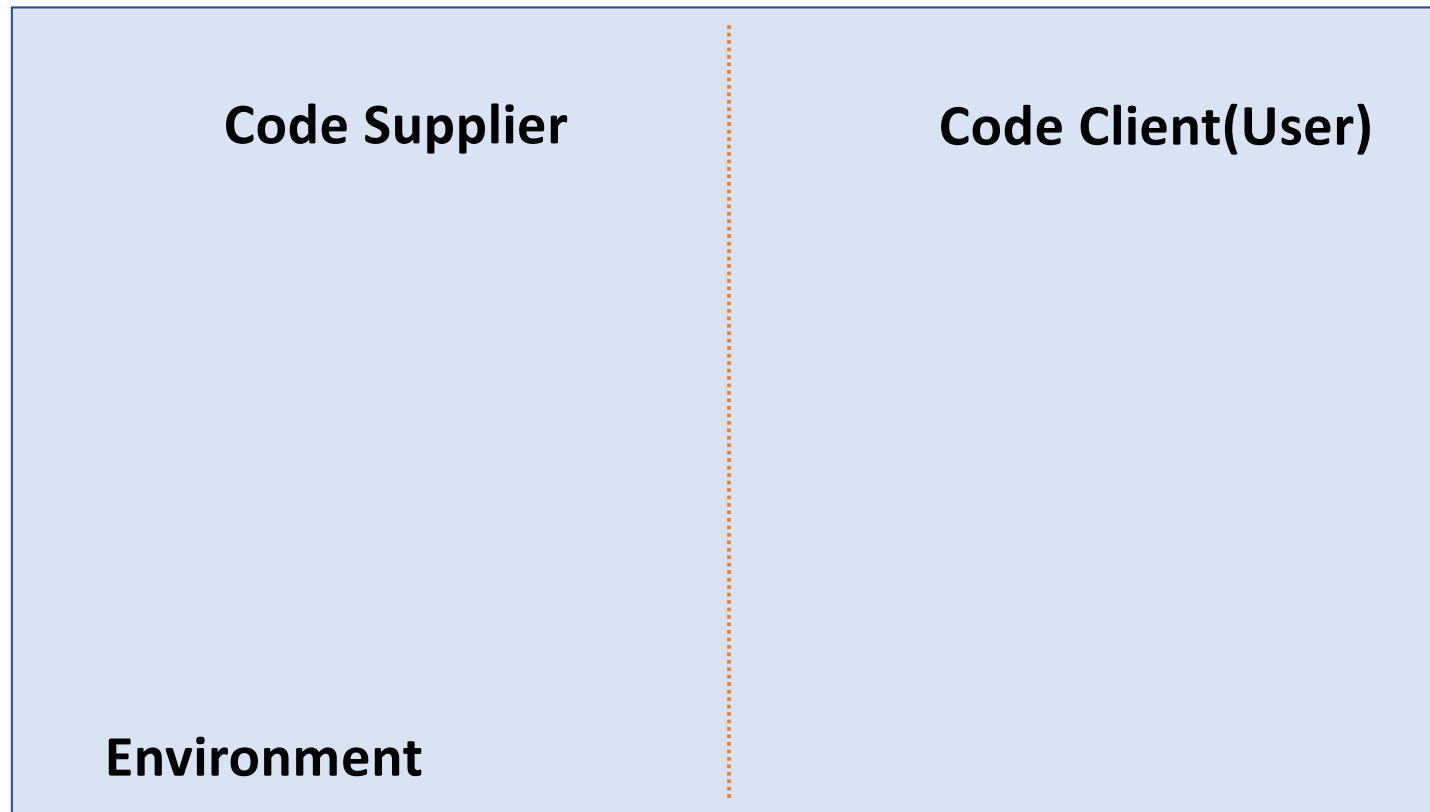


M4 (b) – Design for Robustness

Jin L.C. Guo

This Photo by Unknown Author is licensed under [CC BY-SA](#)

Where can things go wrong?



Java Convention for Checking Preconditions

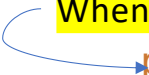
Explicit checks that throw particular, specified exceptions

Use assertion to test a *nonpublic* method's precondition that you believe will be true no matter what a client does with the class.

Java Convention for Private Method

```
* ... ...  
* @pre pStudent != null && !isFull()  
* @post aEnrollment.get(aEnrollment.size()-1) == pStudent()  
*/
```

When this is **private or protected**



```
public void enroll(Student pStudent) {  
    assert pStudent != null && !isFull() : this;  
    aEnrollment.add(pStudent);  
}
```

```
public boolean isFull() {  
    return aEnrollment.size() == aCap;  
}
```

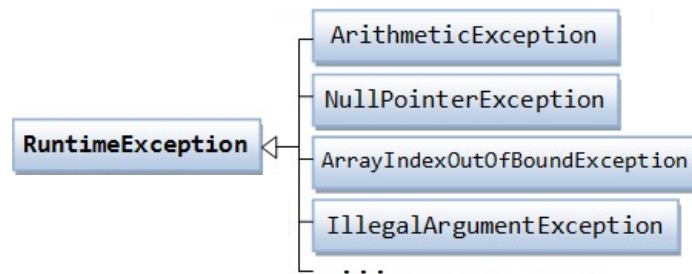
Java Convention for Public Method

```
/**
 * ...
 * @param Student to be enrolled to the Course
 * @throws IllegalStateException if isFull()
 */

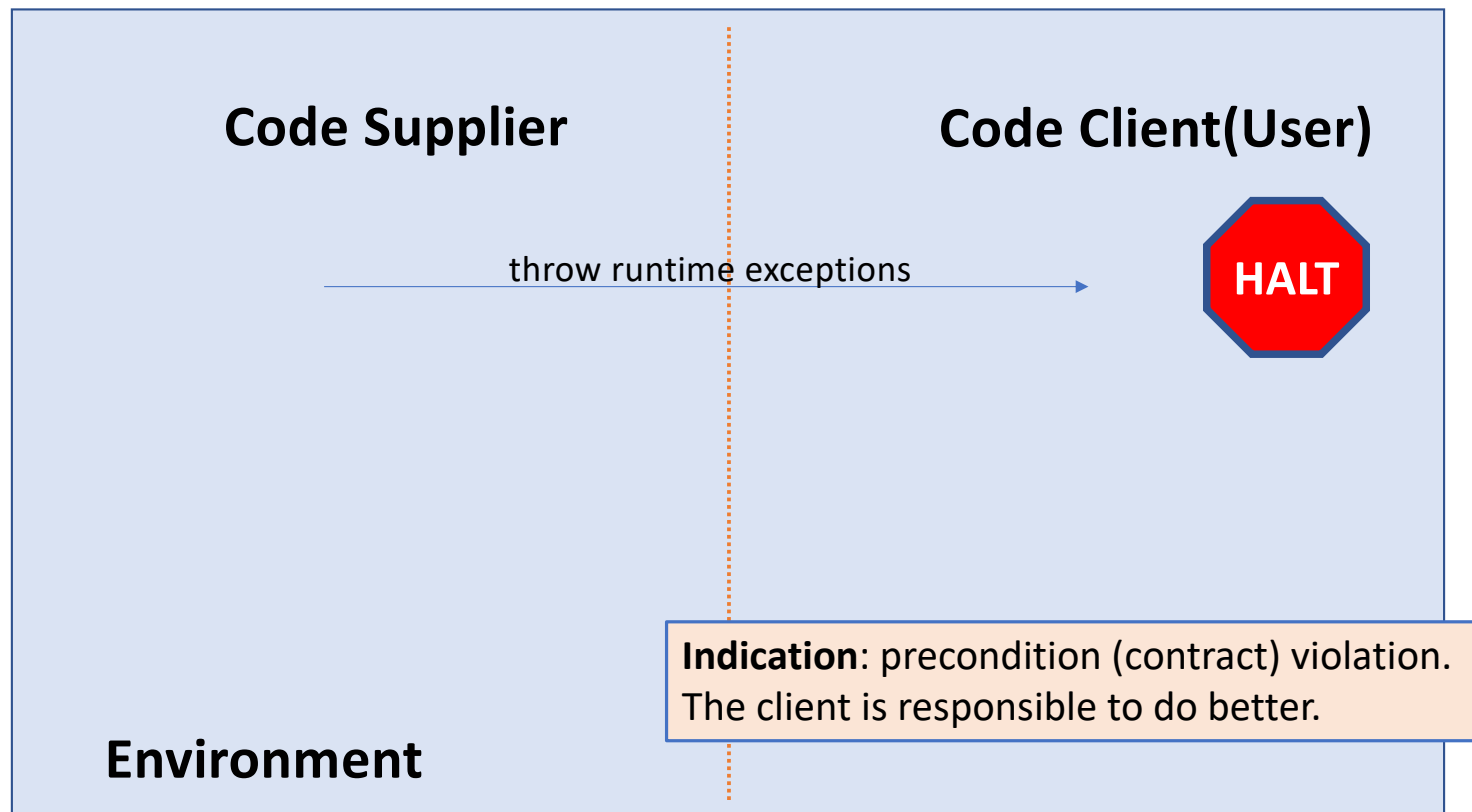
public void enroll(Student pStudent) {

    if (pStudent == null)
        throw new NullPointerException();
    if (isFull())
        throw new IllegalStateException();
    aEnrollment.add(pStudent);
}
```

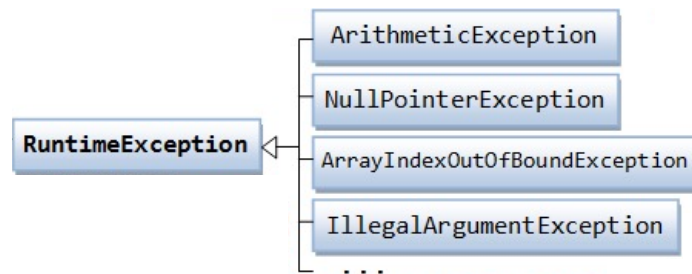
Runtime Exceptions



Code Interaction

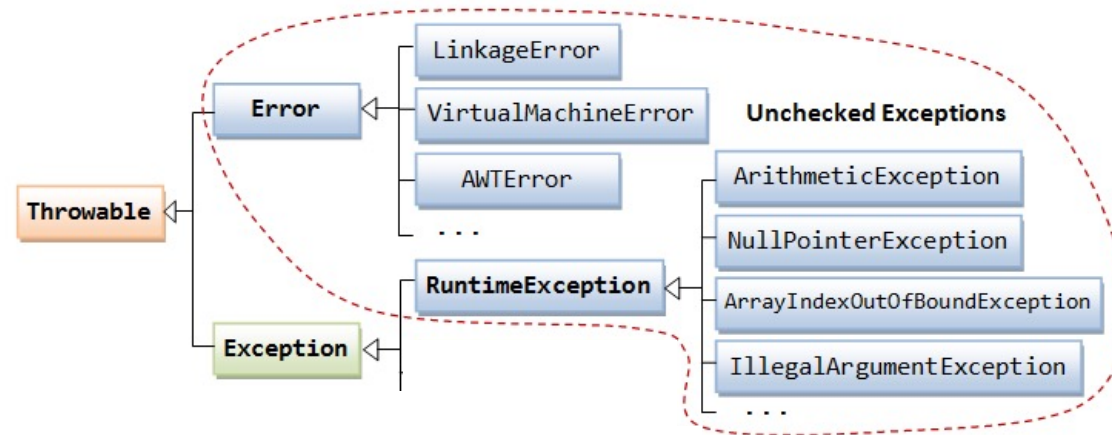


Runtime Exceptions



Unchecked Exceptions

They all cause the program to halt.



The whole hierarchy

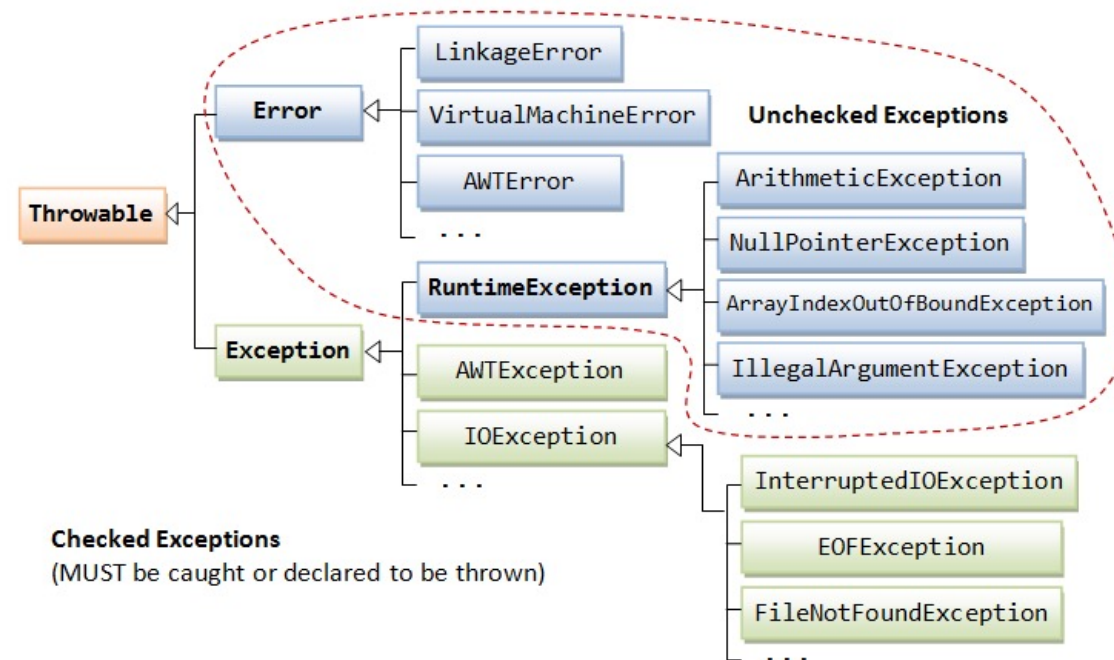
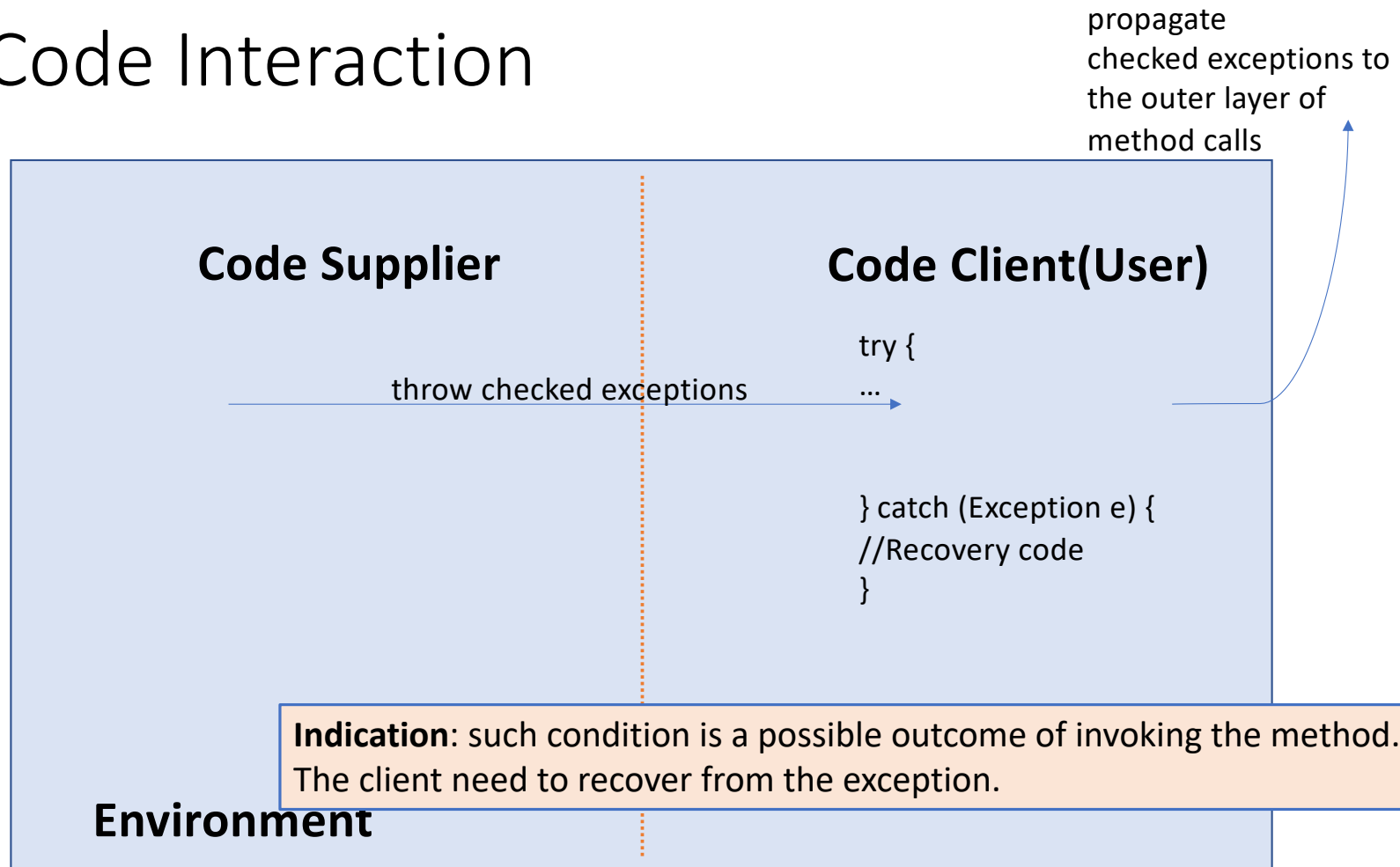


image source: http://www.ntu.edu.sg/home/ehchua/programming/java/images/Exception_Classes.png

Code Interaction



Another design of the `enroll` method

Assume `CourseFullException` is a Checked Exception

```
/**
 * Enroll the student to the course if the course currently is not full
 * @param pStudent to be enrolled to the Course
 * @throws CourseFullException if isFull()
 */
public void enroll(Student pStudent) throws CourseFullException {
    if (pStudent == null)
        throw new NullPointerException();
    if (isFull())
        throw new CourseFullException();
    aEnrollment.add(pStudent);
}
```

Impact to the Client

The client is not obliged to check `isFull()` anymore. However...


```
Course comp303 = new Course("COMP 303", 1);
Undergrad s1 = new Undergrad("00009", "James", "Harris");
Undergrad s2 = new Undergrad("00002", "Benny", "Will");

comp303.enroll(s1);
comp303.enroll(s2);

System.out.println("Done with enrolling students.");
comp303.printEnrolledStudent();
```

Impact to the Client

They have to catch the potential exception or propagate it



```
Course comp303 = new Course("COMP 303", 1);
Undergrad s1 = new Undergrad("00009", "James", "Harris");
Undergrad s2 = new Undergrad("00002", "Benny", "Will");
try {
    comp303.enroll(s1);
    comp303.enroll(s2);
    System.out.println("Done with enrolling students.");
} catch (CourseFullException e){
    ... // Handle the exception
    e.printStackTrace();
}
comp303.printEnrolledStudent();
```

Summary: Checked vs Unchecked Exception

- Checked Exceptions

Code supplier needs to declare in the method signature.

Code client needs to catch or declare.

Used for abnormal cases but can be recovered at runtime

- Unchecked Exceptions

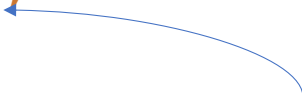
Code supplier does **not** have to declare it

Code client does **not** have to catch nor declare it.

Used for programming errors or things should not happen at runtime.

Any problem with this method?

```
public void writeToFile(Course pCourse, String pFilePath) {  
    File file = new File(pFilePath);  
  
    try {  
        FileWriter fileWriter = new FileWriter(file);  
        for (Student s : pCourse) {  
            fileWriter.write(s.toString());  
            fileWriter.write("\n");  
        }  
        fileWriter.close();  
    } catch (IOException e) {  
        e.printStackTrace();  
    }  
}
```



If exceptions happen here

The `final` block

```
public void writeToFile(Course pCourse, String pFilePath) {  
    File file = new File(pFilePath);  
    FileWriter fileWriter = null;  
    try {  
        fileWriter = new FileWriter(file);  
        for (Student s : pCourse) {  
            fileWriter.write(s.toString());  
            fileWriter.write("\n");  
        }  
    } catch (IOException e) {  
        e.printStackTrace();  
    } finally {  
        try {  
            fileWriter.close();  
        } catch (IOException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

Alternative: try-with-Resources statement

```
public void writeToFile2(Course pCourse, String pFilePath) {  
    File file = new File(pFilePath);  
    try (FileWriter fileWriter = new FileWriter(file)){  
        for (Student s : pCourse) {  
            fileWriter.write(s.toString());  
            fileWriter.write("\n");  
        }  
    } catch (IOException e) {  
        e.printStackTrace();  
    }  
}
```

`close()` will be called when the try block exits.

Case study:

```
if(!comp303.isFull())  
    comp303.enroll(s2);
```

VS

```
try {  
    comp303.enroll(s2);  
} catch (CourseFullException e){  
    ... // Handle the exception  
}
```

When Not to use Exceptions

- For ordinary control flow