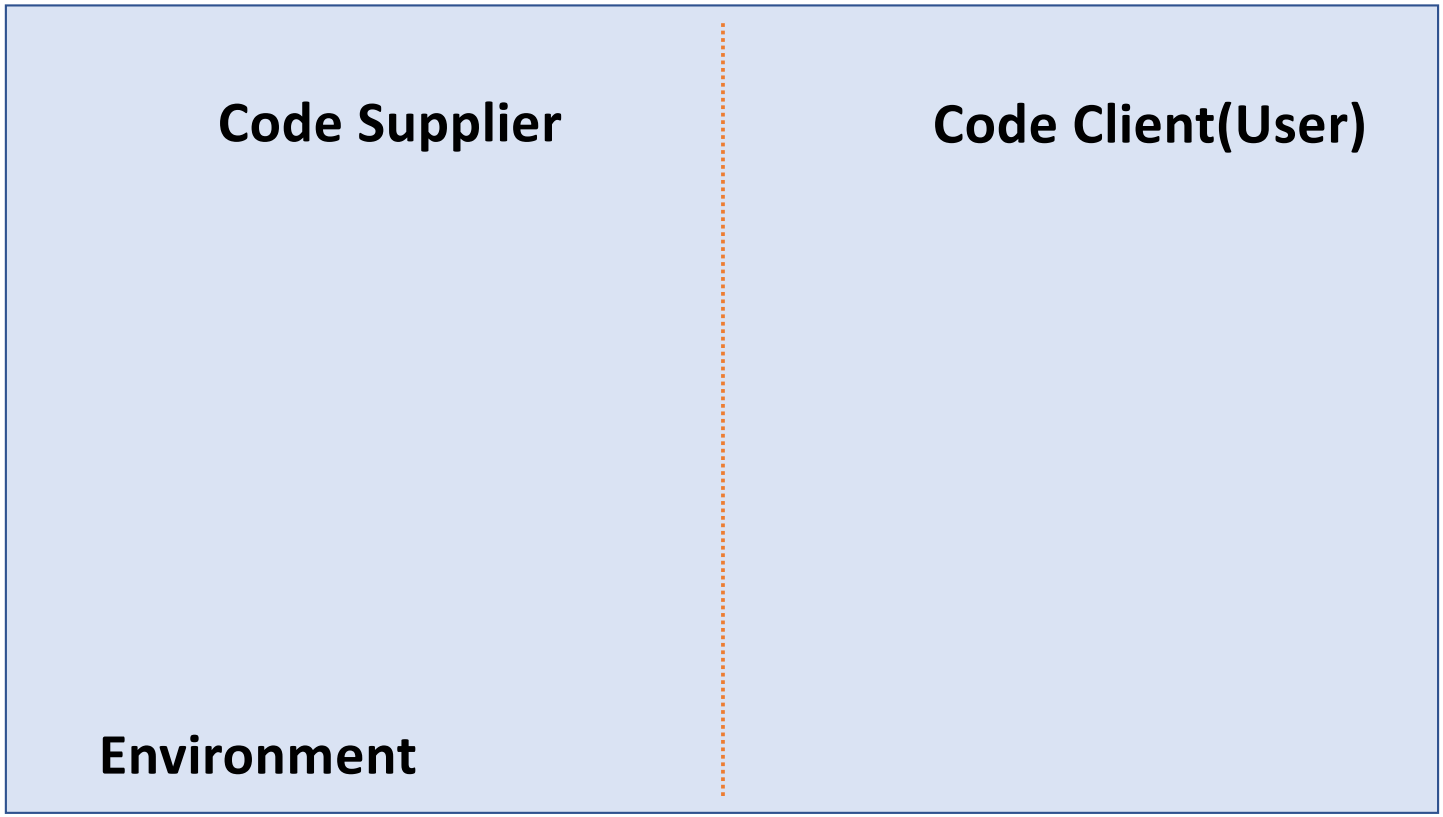
M4 (b) – Design for Robustness

Jin L.C. Guo

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;
}
```

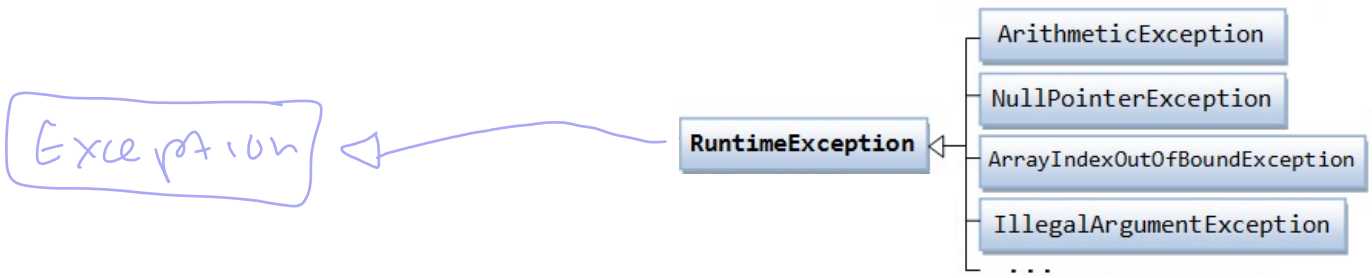
Java Convention for Public Method

```
/**
 * … …
 * @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

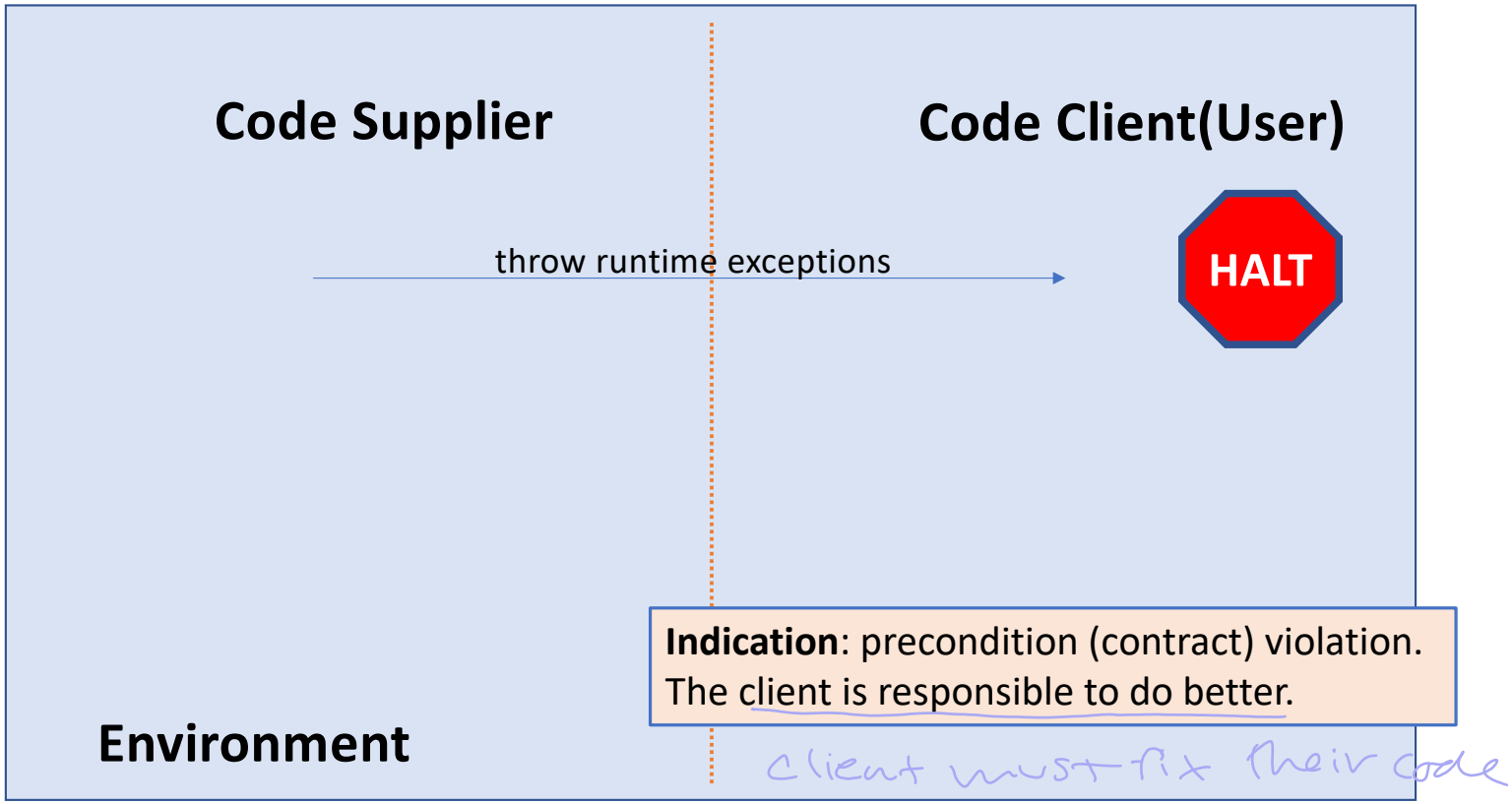
Runtime Exceptions

unchecked exceptions



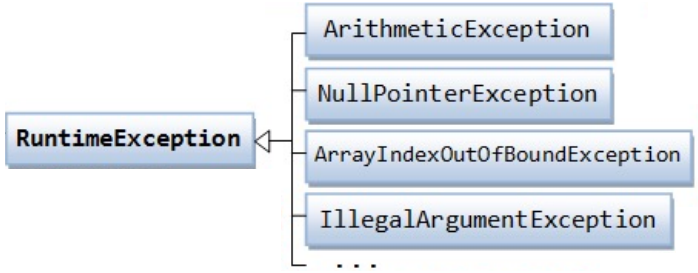
Code Interaction

runtime exceptions
need to be taken
care of / program needs
to stop



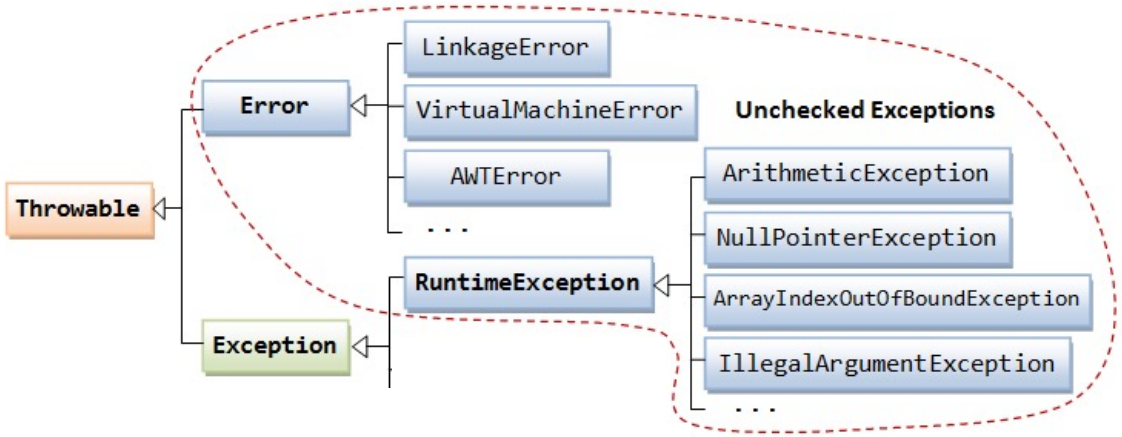
client must fix their code

Runtime Exceptions



Unchecked Excaptions

They all cause the program to halt.



The whole hierarchy

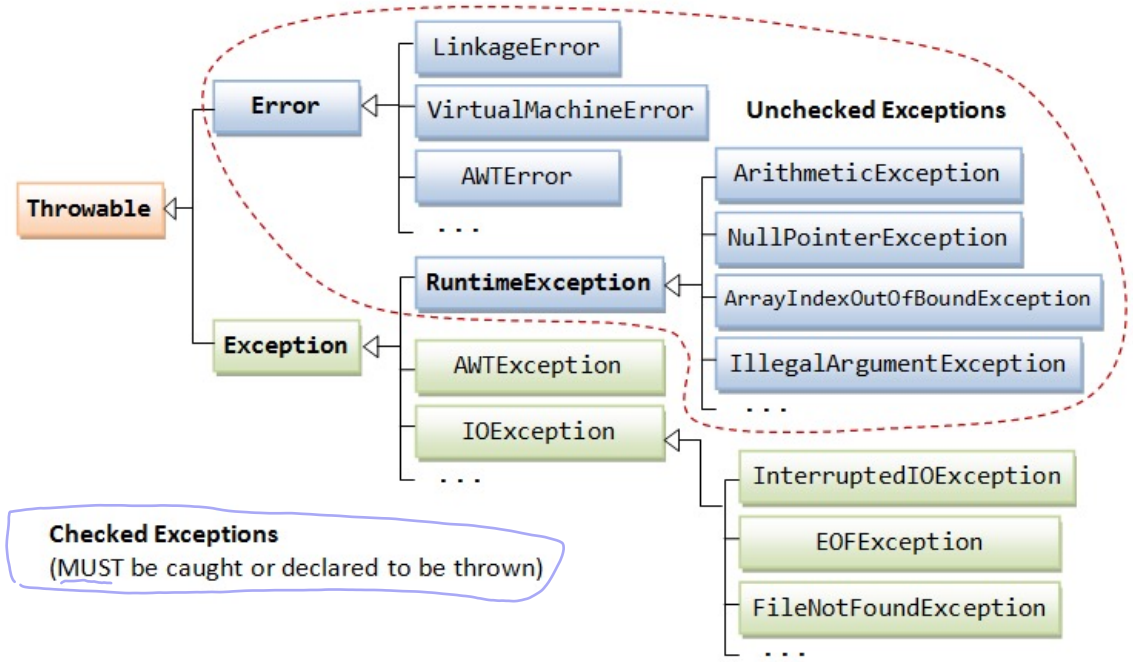


image source:http://www.ntu.edu.sg/home/ehchua/programming/java/images/Exception_Classes.png

Code Interaction

propagate checked exceptions to the outer layer of method calls

outer client

Code Supplier

throw checked exceptions

Code Client(User)

```
try {
...
```

call method

```
} catch (Exception e) {
//Recovery code
}
```

Indication: such condition is a possible outcome of invoking the method. The client need to recover from the exception.

Environment

checked exception — client can use a try catch block

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);
}
```

- only option

- define a new checked exception

Client has to handle error

isFull() is not a precondition anymore

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

eg
use try catch

- 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.

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

doesn't close if there an exception

The final block

allocate memory in try → free it in finally

```
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();
        }
    }
}
```

happens if there an exception or not

nested try catch

try (argument)

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();
    }
}
```

resources allocation

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

dont need to do it manually

Case study:

design by contract

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

assumes client reads documentation

could happen often, program stops often

vs

environment handling

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

When Not to use Exceptions

- For ordinary control flow