



Xi'an Jiaotong-Liverpool University

西交利物浦大學

# CPT204 Advanced Object-Oriented Programming

## Lecture 6

### Exception, Linked List 3

# Welcome !

---

- Welcome to Lecture 6 !
- In this lecture we are going to
  - learn about null values
  - learn about exception
    - checked vs unchecked
    - handling exception
  - continue improving our homemade linked list, while learning about
    - copy constructor
    - shallow copy vs deep copy

# Null References (1)

---

- In Java, references to objects and arrays can also take on the special value **null**, which means that the reference doesn't point to an object
  - Null values are an unfortunate hole in Java's type system
- Primitives cannot be null and the compiler will reject such attempts with **static errors**:

```
int size = null;      // illegal
double depth = null; // illegal
```

## Null References (2)

---

- On the other hand, we can assign `null` to any non-primitive variable and the compiler happily accepts this code at compile time:

```
String name = null;  
int[] points = null;
```

- But you'll get **errors at runtime** because you can't call any methods or use any fields with one of these references:

```
name.length()    // throws NullPointerException  
points.length    // throws NullPointerException
```

## Null References (3)

---

- Note that `null` is **not** the same as an empty string `""` or an empty array
  - On an empty string or empty array, you **can** call methods and access fields
  - The length of an empty array or an empty string is 0
  - The length of a string variable that points to null isn't anything: calling `length()` throws a `NullPointerException`
- Also note that arrays of non-primitives and collections like `List` might be non-null but contain `null` as a value:

```
String[] names = new String[] { null };  
List<Double> sizes = new ArrayList<>();  
sizes.add(null);
```

- These nulls are likely to cause errors as soon as someone tries to use the contents of the collection!

# Null Values (1)

---

- Null values are troublesome and unsafe, and in fact in most good Java programming, null values are ***implicitly disallowed*** in *parameters* and *return values*
  - so every method implicitly has a precondition on its object and array parameters that they be non-null, and
  - every method that returns an object or an array implicitly has a postcondition that its return value is non-null
  - if a method allows null values for a parameter, it should ***explicitly*** state it, or if it might return a null value as a result, it should ***explicitly*** state it

## Null Values (2)

---

- There are extensions to Java that allow you to forbid null directly in the type declaration:

```
static boolean addAll(@NonNull List<T> list1, @NonNull List<T> list2)
```

where it can be checked automatically at compile time or runtime

- Google has their own discussion of null in Guava, the company's core Java libraries. The project explains:

Careless use of `null` can cause a staggering variety of bugs. Studying the Google code base, we found that something like 95% of collections weren't supposed to have any null values in them, and having those **fail fast** rather than silently accept `null` would have been helpful to developers.

Additionally, `null` is unpleasantly ambiguous. It's rarely obvious what a `null` return value is supposed to mean — for example, `Map.get(key)` can return `null` either because the value in the map is `null`, or the value is not in the map. Null can mean failure, can mean success, can mean almost anything. Using something other than `null` **makes your meaning clear**.



# Exceptions

---

- Next, we discuss how to handle *exceptional cases* in a way that is safe from bugs and easy to understand
- A method's signature — its name, parameter types, return type — is a core part of its specification, and the signature may also include *exceptions* that the method may *trigger*

# Exceptions for Signaling Bugs

---

- We have already seen some exceptions in our Java programming so far, such as
  - `IndexOutOfBoundsException`, thrown when a list index `list.get(i)` is outside the valid range for the list `list`
  - `NullPointerException`, thrown when trying to call a method on a `null` object reference
- These exceptions generally *indicate bugs in your code*, and the information displayed by Java when the exception is thrown can help you find and fix the bug
- `IndexOutOfBoundsException` and `NullPointerException` are probably the most common exceptions of this sort
- Other examples include:
  - `ArithmeticException`, thrown for arithmetic errors like integer division by zero
  - `NumberFormatException`, thrown by methods like `Integer.parseInt` if you pass in a string that cannot be parsed into an integer

# Exceptions for Special Results (1)

---

- Exceptions are not just for signaling bugs
- They can be used to improve the structure of code that involves procedures with **special results**
- An unfortunately common way to handle special results is to return special values
- Lookup operations in the Java library are often designed like this:  
you get an index of -1 when expecting a positive integer, or a null reference when expecting an object
- This approach is OK if used sparingly, but it has two problems :
  - First, it's tedious to check the return value
  - Second, it's easy to forget to do it

## Exceptions for Special Results (2)

---

- Also, it's not always easy to find *a special value*.

Suppose we have a BirthdayBook class with a lookup method.

Here's one possible method signature:

```
class BirthdayBook {  
    LocalDate lookup(String name) { ... }  
}
```

(LocalDate is part of the Java API)

- What should the method do if the birthday book *doesn't* have an entry for the person whose name is given?
  - we could return some *special date* that is not going to be used as a real date
  - bad programmers have been doing this for decades; they would return 9/9/99, for example, since it was obvious that no program written in 1960 would still be running at the end of the century

btw, they are wrong,  
caused Y2K bug

## Exceptions for Special Results (3)

---

- Here's a better approach: the method *throws an exception*:

```
LocalDate lookup(String name) throws NotFoundException {  
    ...  
    if ( ...not found... )  
        throw new NotFoundException();  
    ...  
}
```

- The caller *handles* the exception with *a catch clause*, for example:

```
BirthdayBook birthdays = ...  
try {  
    LocalDate birthdate = birthdays.lookup("Alyssa");  
    // we know Alyssa's birthday  
} catch (NotFoundException nfe) {  
    // her birthday was not in the birthday book  
}
```

- Now there's no need for any special value, nor the checking associated with it

## Exceptions for Special Results (3)

- Here's a better approach: the method *throws an exception*:

```
LocalDate lookup(String name) throws NotFoundException {  
    ...  
    if ( ...not found... )  
        throw new NotFoundException();  
    ...  
}
```

advertise the exception  
in method's signature

the condition that triggers throwing exc

an **object** of type NotFoundException  
is thrown by lookup if date not found,  
method lookup execution then ends

- The caller *handles* the exception with *a catch clause*, for example:

```
BirthdayBook birthdays = ...  
try {  
    LocalDate birthdate = birthdays.lookup("Alyssa");  
    // we know Alyssa's birthday  
} catch (NotFoundException nfe) {  
    // her birthday was not in the birthday book  
}
```

method that throws exception  
is called inside try block here

codes here executed if lookup does  
not throw a NotFoundException object

codes here executed if lookup throws  
a NotFoundException object

- Now there's no need for any special value, nor the checking associated with it

# In-Class Quiz 1

---

- We use BirthdayBook with the lookup method that throws NotFoundException
- Assume we have initialized the birthdays variable to point to a BirthdayBook, and that "Makima" is **not** in that birthday book
- What will happen with the following code:

```
try {  
    LocalDate birthdate = birthdays.lookup("Makima");  
}  
System.out.println("done");
```

- static error caused by incorrect syntax
- static error caused by undeclared variable
- dynamic error caused by NotFoundException
- no errors and it prints "done"

## In-Class Quiz 2

---

- We use BirthdayBook with the lookup method that throws NotFoundException
- Assume we have initialized the birthdays variable to point to a BirthdayBook, and that "Makima" is **not** in that birthday book
- What will happen with the following code:

```
try {  
    LocalDate birthdate = birthdays.lookup("Makima");  
} catch (NotFoundException nfe) {  
    birthdate = LocalDate.now();  
}  
System.out.println("done");
```

- ☐ static error caused by incorrect syntax
- ☐ static error caused by undeclared variable
- ☐ dynamic error caused by NotFoundException
- ☐ no errors and it prints "done"



## In-Class Quiz 3

---

- We use BirthdayBook with the lookup method that throws NotFoundException
- Assume we have initialized the birthdays variable to point to a BirthdayBook, and that "Makima" is **not** in that birthday book
- What will happen with the following code:

```
try {  
    LocalDate birthdate = birthdays.lookup("Makima");  
} catch (Exception NotFoundException) {  
}  
System.out.println("done");
```

- static error caused by incorrect syntax
- static error caused by undeclared variable
- dynamic error caused by NotFoundException
- no errors and it prints "done"

# Exception Message (1)

---

- All exceptions may have a message associated with them in the constructor
  - to pass useful information about the cause of throwing the exception

```
LocalDate lookup(String name) throws NotFoundException {  
    ...  
    if ( ...not found... )  
        throw new NotFoundException(name + "not found");  
    ...  
}
```

- in the catch block, it is accessed by getMessage()

```
try {  
    LocalDate birthdate = birthdays.lookup("Makima");  
} catch (NotFoundException nfe) {  
    System.err.println("birthdate of " + nfe.getMessage());  
}
```

## Exception Message (2)

---

- All exceptions may have a message associated with them in the constructor
  - if **not** provided in the constructor, the reference to the message string is `null`
  - this can result in confusing stack traces that start, for example:

```
birthday.NotFoundException: null  
at birthday.BirthdayBook.lookup(BirthdayBook.java:42)
```

- the `null` is misleading : in this case, it tells you the `NotFoundException` had no message string, **not that** a null value was responsible for the exception

# Checked and Unchecked Exceptions (1)

---

- We've seen two different purposes for exceptions:
  - special results, and
  - bug detection
- As a general rule, you'll want to use **checked exceptions** to signal *special results* and **unchecked exceptions** to *signal bugs*

## Checked and Unchecked Exceptions (2)

---

- **Checked exceptions** are called that because they are *checked by the compiler*:
  - if a method might throw a checked exception, the possibility must be declared in its signature
    - `NotFoundException` would be a checked exception, and that's why the signature ends with `throws NotFoundException`
  - if a method calls another method that may throw a checked exception, it must either handle it (with try-catch), **or** declare the exception itself, since if it isn't caught locally it will be propagated up to callers
- So if you call `BirthdayBook`'s `lookup` method and *forget to handle* the `NotFoundException`, the compiler will **reject** your code
  - this is very useful, because it ensures that exceptions that are expected to occur will be handled

## Checked and Unchecked Exceptions (3)

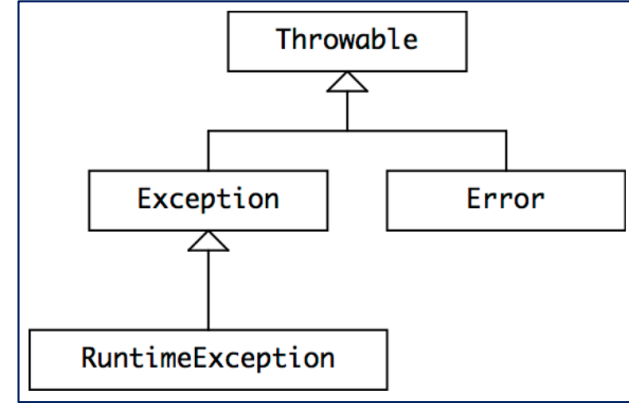
---

- ***Unchecked exceptions***, in contrast, are used to signal bugs
  - these exceptions are **not** expected to be handled by the code, except perhaps at the top level
  - we wouldn't want every method up the call chain to have to declare that it (might) throw all the kinds of bug-related exceptions that can happen at lower call levels: index out of bounds, null pointers, illegal arguments, assertion failures, etc
- As a result, for an unchecked exception the compiler will **not** check for try-catch or a throws declaration
  - Java *still allows* you to write a throws clause for an unchecked exception as part of a method signature, but this has ***no effect***, we don't recommend doing it

# Java Exception Hierarchy

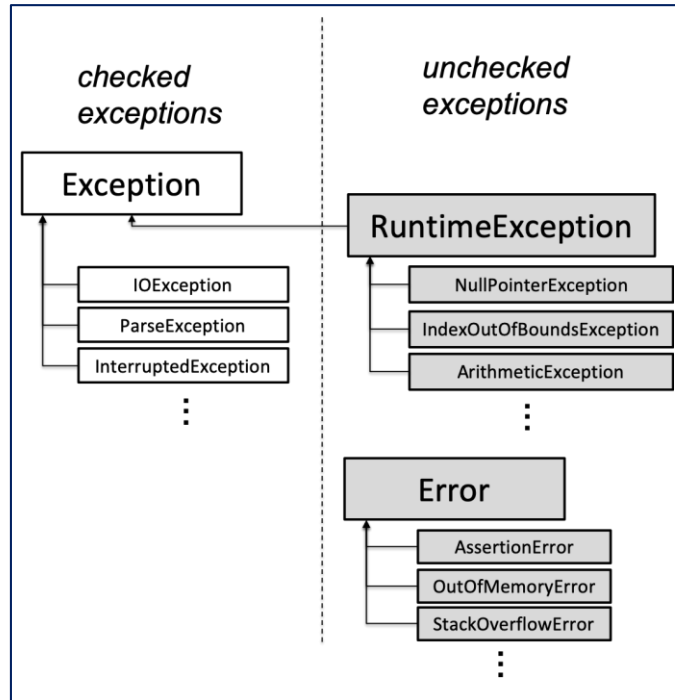
---

- To understand how Java decides whether an exception is checked or unchecked, let's look at the class hierarchy for Java exceptions
- **Throwable** is the class of objects that can be thrown or caught
  - Throwable's implementation records a stack trace at the point where the exception was thrown, along with an optional string describing the exception
  - any object used in a throw or catch statement, or declared in the throws clause of a method, must be a subclass of Throwable



# Checked vs Unchecked Exception in Java (1)

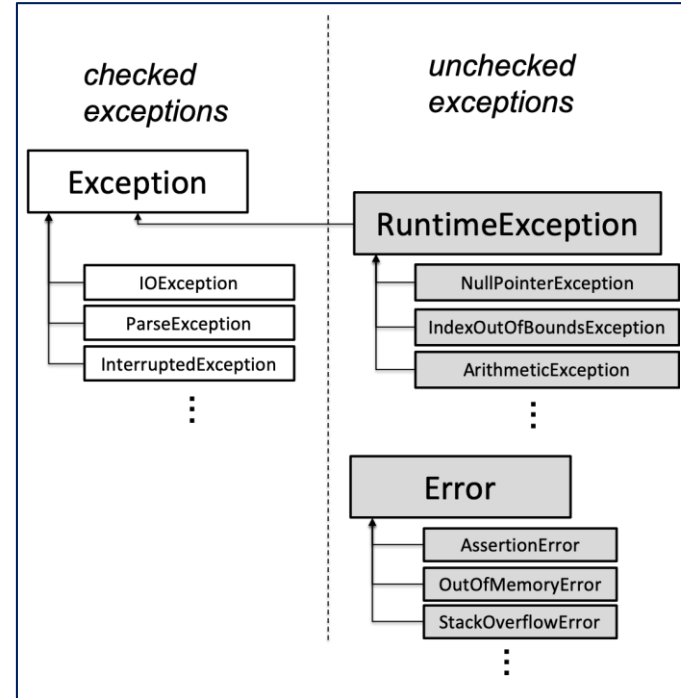
- **Exception** is the normal base class of **checked exceptions**
- The compiler applies **static checking** to methods using these exceptions
  - A checked exception **must** either be caught or declared when it's possible for it to be thrown





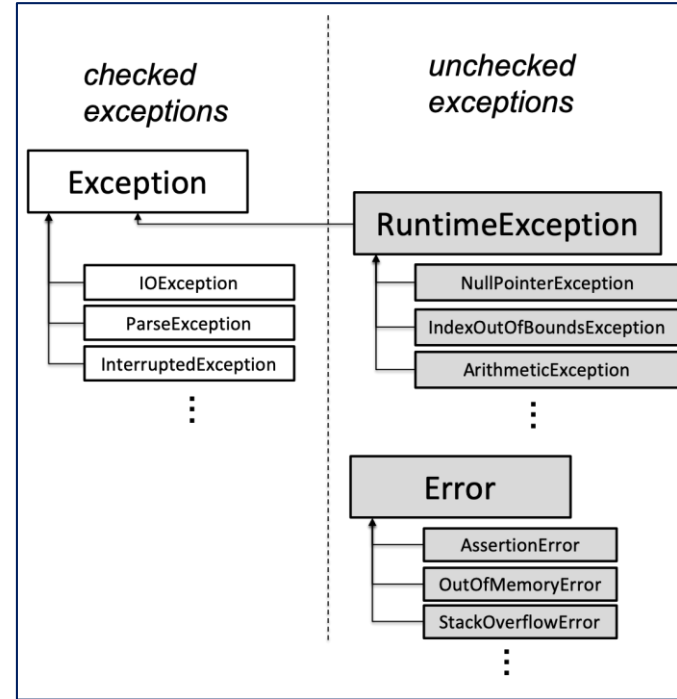
# Checked vs Unchecked Exception in Java (2)

- However, `RuntimeException` and its subclasses are **unchecked exceptions**
- `RuntimeException` and its subclasses ***don't have to*** be declared in the throws clause of a method that throws them,
  - and ***don't have to*** be caught or declared by a caller of such a method



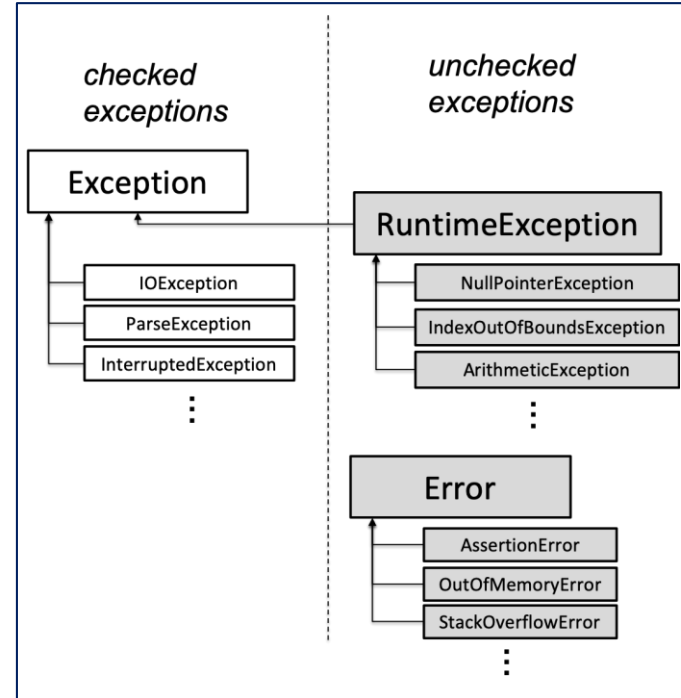
# Checked vs Unchecked Exception in Java (3)

- In addition, **Error** and its subclasses are also **unchecked exceptions**
- This part of the hierarchy is reserved for errors produced by the Java runtime system, such as `StackOverflowError` and `OutOfMemoryError`
- For some reason `AssertionError` also extends `Error`, even though it indicates a bug in user code, not in the runtime
- Errors should be considered unrecoverable, and should ***not*** be caught by your code



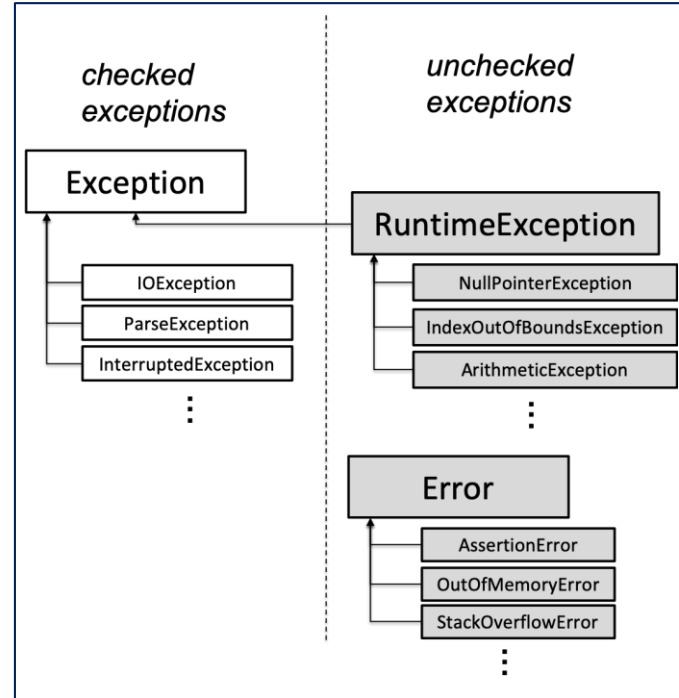
# Creating Exception

- When you define your own exceptions, ***you should either subclass Exception to make it an checked exception,***
  - ***or subclass RuntimeException to make it unchecked exception***
- ***Don't*** subclass Error or Throwable, because these are reserved by Java itself



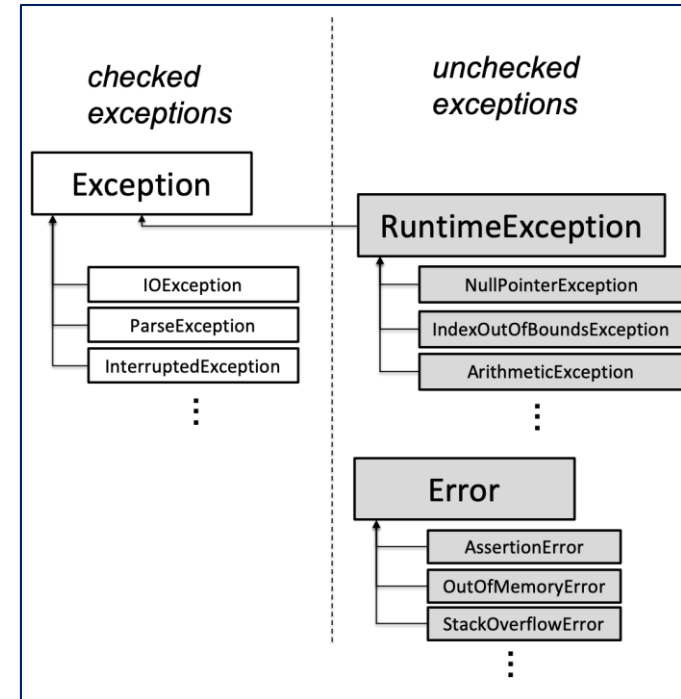
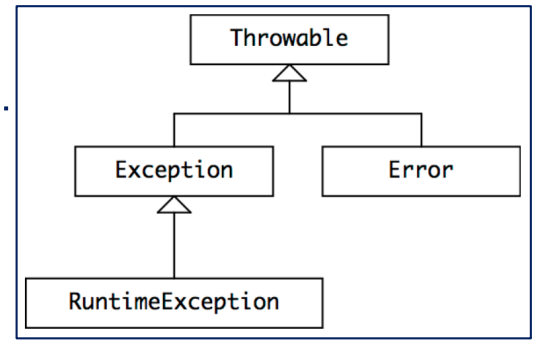
# Catching Exception

- When you catch an exception with a try-catch statement, you should catch the most specific exception class possible
- If you are expecting a `FileNotFoundException`, that's what your catch statement should use
- Catching a broad class of exceptions, like `Exception` or `RuntimeException` or `Error`, is not safe from bugs or ready for change
  - because it catches **every** possible subclass of these exceptions, which may interfere with static checking and hide bugs



# Java Exception Hierarchy

- One of the *confusing aspects* of the Java exception hierarchy is that `RuntimeException` is itself a subclass of `Exception`
- So the whole Exception family includes **both** *checked* exceptions (its direct descendants) **and** *unchecked* exceptions (the `RuntimeException` branch of the family)
- But `Error` is **not** a subclass of `Exception`, so all the *unchecked* Error-like exceptions are outside the Exception family



# Exception Design Considerations (1)

---

- The rule we have given — use checked exceptions for special results (i.e., anticipated situations), and unchecked exceptions to signal bugs (unexpected failures) — makes sense, but it isn't the end of the story
- Exceptions in Java aren't as lightweight as they might be
  - They are expensive!
- Aside from the performance penalty, exceptions in Java incur another (more serious) cost: they're a pain to use, in both method design and method use
  - If you design a method to have its own (new) exception, you have to create a new class for the exception
  - If you call a method that can throw a checked exception, you have to wrap it in a try-catch statement (*even* if you know the exception will never be thrown)

## Exception Design Considerations (2)

---

- Suppose, for example, you're designing a queue
  - Should *popping* the queue throw a checked exception when the queue is empty?
  - Suppose you want to support a style of programming in the client in which the queue is popped in a loop, until the exception is thrown
  - So you choose *a checked exception*
  - Now some client wants to use the method in a context in which, immediately prior to popping, the client *tests* whether the queue is empty and *only pops if it isn't*
  - That client will ***still need*** to wrap the call in a try-catch statement!

## Exception Design Considerations (3)

---

- This suggests a more refined rule:
  - You should use an ***unchecked exception*** only to signal an *unexpected failure* (i.e. a bug),  
**or** if you *expect* that clients will *usually write code that ensures the exception will not happen*, because there is a convenient and inexpensive way to avoid the exception
  - Otherwise you should use a ***checked exception***



## Exception Design Considerations (4)

---

- Here are some examples of applying this rule :
  - `Queue.pop()` throws an ***unchecked*** `EmptyQueueException` when the queue is empty
    - because it's reasonable to expect the caller to *avoid* this with a call like `Queue.size()` or `Queue.isEmpty()`
  - `Url.getWebPage()` throws a ***checked*** `IOException` when it can't retrieve the web page
    - because it's not easy for the caller to *prevent* this

## Exception Design Considerations (5)

---

- `int integerSquareRoot(int x)` throws a ***checked*** `NotPerfectSquareException` when `x` has no integral square root,
  - because testing whether `x` is a perfect square is *just as hard* as finding the actual square root, so it's *not reasonable* to expect the caller to *prevent* it
- The cost of using exceptions in Java is one reason that many Java API's use the `null` reference as a special value
  - it's not a terrible thing to do, so long as it's done judiciously, and carefully specified

## Declare Exceptions in Specification (1)

---

- Since an exception is a possible output from a method, it may have to be described in the postcondition for the method
- The Java way of documenting an exception as a postcondition is a `@throws` clause in the Javadoc comment
  - Recall that Java may also require the exception to be included in the method signature, using a throws declaration
- Exceptions that **signal a special result** (checked or unchecked) are always documented with a Javadoc `@throws` clause, specifying the conditions under which that special result occurs

## Declare Exceptions in Specification (2)

---

- For a **checked exception**, Java also requires the exception to be mentioned in a throws declaration in the method signature
- For example, suppose `NotPerfectSquareException` were a checked exception; You would need to mention it in both `@throws` in the Javadoc and `throws` in the method signature:

```
/**  
 * Compute the integer square root.  
 * @param x value to take square root of  
 * @return square root of x  
 * @throws NotPerfectSquareException if x is not a perfect square  
 */  
int integerSquareRoot(int x) throws NotPerfectSquareException
```

## Declare Exceptions in Specification (3)

---

- For an **unchecked exception** that signals a special result, Java allows but doesn't require the throws clause; but it is better to omit the exception from the throws clause in this case, to avoid misleading a human programmer into thinking that the exception is checked
- For example, suppose you defined EmptyQueueException as an unchecked exception, then you should document it with @throws, but **not** include it in the method signature:

```
/**  
 * Pops a value from this queue.  
 * @return next value in the queue, and removes the value from the queue  
 * @throws EmptyQueueException if this queue is empty  
 */  
int pop()
```

## Declare Exceptions in Specification (4)

---

- Exceptions that **signal unexpected failures** – bugs in either the client or the implementation – are not part of the postcondition of a method, so they should **not** appear in either `@throws` or `throws`
- For example, `NullPointerException` *need never be* mentioned in a spec
  - an implicit precondition already disallows null values, which means that a valid implementation is free to throw it without any warning if a client ever passes a null value
- So this spec, for example, *never* mentions `NullPointerException`:

```
/**  
 * @param list list of strings to convert to lower case  
 * @return new list t, same length as list,  
 *         where t[i] is list[i] converted to lowercase for all valid indices i  
 */  
static List<String> toLowerCase(List<String> list)
```

# Abuse of Exceptions (1)

---

- Consider the following example from Effective Java, Joshua Bloch:

```
try {  
    int i = 0;  
    while (true)  
        a[i++].f();  
} catch (ArrayIndexOutOfBoundsException e) { }
```

- What does this code do?

It is not at all obvious from inspection, and that's reason enough not to use it

- The infinite loop terminates by throwing, catching, and ignoring an `ArrayIndexOutOfBoundsException` when it attempts to access the first array element outside the bounds of the array

## Abuse of Exceptions (2)

---

- It is supposed to be equivalent to:

```
for (int i = 0; i < a.length; i++) {  
    a[i].f();  
}
```

- Or (using appropriate type T) to:

```
for (T x : a) {  
    x.f();  
}
```

- The exception-based idiom, Bloch writes:
  - ... is a misguided attempt to improve performance based on the **faulty reasoning** that, since the VM checks the bounds of array accesses, the normal loop termination test (`i < a.length`) is redundant and should be avoided



## Abuse of Exceptions (3)

---

- However, because exceptions in Java are designed for use only under exceptional circumstances, few, if any, JVM implementations attempt to optimize their performance
  - On a typical machine, the exception-based idiom runs 70 times slower than the standard one when looping from 0 to 99
- Much worse than that, the exception-based idiom is not even guaranteed to work!
  - Suppose the computation of `f()` in the body of the loop contains a bug that results in an out-of-bounds access to some unrelated array
  - What happens?
  - If a *reasonable loop idiom* were used, the bug would generate an uncaught exception, resulting in immediate thread termination with a full stack trace
  - If the *misguided exception-based loop* were used, the bug-related exception would be caught and misinterpreted as a normal loop termination!

# Copy Constructor

---

- Copy constructor is a special constructor that takes as input another object of the same type and create a new object that is a copy of the input
  - the other object is usually called other
  - a copy here for a data structure usually means having the same elements
- Let us create a copy constructor for our running example, `SLList<T>`

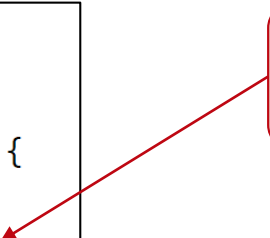
# Shallow Copy

---

- Suppose you code the Copy Constructor for SLList<T> as below

```
// Shallow Copy  
// NOT what we want !  
public SLList(SLList<T> other) {  
    sentinel = other.sentinel;  
    size = other.size;  
}
```

you simply set the sentinel to point to other's sentinel and copy the size



- This is called **shallow copy** or **aliasing**
- Not what we want, because we want to create a new copy
  - while this is just having a new pointer to the old object, other
  - viewed another way, this is just giving the old object a new name / alias

# Deep Copy

---

- What we want is to create an entirely new `SLList<T>`, with the exact same items as other
- This is called **deep copy**
  - The input and the copy output should be different objects
  - If you change other, the new `SLList<T>` you created should **not** change as well

# SLList<T> Copy Constructor

---

- Here is the copy constructor for SLList<T> that correctly uses deep copy

```
/** Creates a (deep) copy SLList of other. */
public SLList(SLList<T> other) {
    sentinel = new Node(i: null, n: null);
    size = 0;

    Node p = sentinel;
    Node q = other.sentinel;
    while (q.next != null) {
        p.next = new Node(q.next.item, n: null);
        p = p.next;
        q = q.next;
        size += 1;
    }
}
```

start with an empty list

create a new node and  
copy from other one-by-one

## SLList<T> Copy Constructor 2

---

- Another way is to use methods that we have previously defined!

```
/** Creates a (deep) copy SLList of other. */  
public SLList(SLList<T> other) {  
    this();  
  
    Node q = other.sentinel;  
    while (q.next != null) {  
        this.addLast(q.next.item);  
        q = q.next;  
    }  
}
```

call the empty constructor

use addLast to add the item  
from other one-by-one

# NullPointerException

---

- If we try to copy a null, we will get a NullPointerException

```
// NullPointerException  
SLList<String> listStr1 = null;  
SLList<String> listStr2 = new SLList<>(listStr1);  
listStr2.printList();
```

NullPointerException  
thrown when it tries to  
access sentinel of null

# Unchecked IllegalArgumentException

---

- We can avoid that by using an unchecked exception `IllegalArgumentException`

```
/**
 * Creates a deep copy SLList of other.
 * @param other an SLList object
 * @throws IllegalArgumentException if other is null
 */
public SLList(SLList<T> other) {

    if (other == null) {
        throw new IllegalArgumentException("other is null");

    }

    sentinel = new Node(i: null, n: null);
    size = 0;

    Node p = sentinel;
    Node q = other.sentinel;
    while (q.next != null) {
        this.addLast(q.next.item);
        q = q.next;
    }
}
```

no need to advertise

check the throwing condition

pass a message



# Catching IllegalArgumentException

---

- The exception can be caught in main
  - note that without try-catch the code still compiles, since we use an unchecked exception

```
// Try-Catch IllegalArgumentException  
SLList<String> listStr1 = null;  
try {  
    SLList<String> listStr2 = new SLList<>(listStr1);  
} catch (IllegalArgumentException iae) {  
    System.out.println("Copy failed, " + iae.getMessage());  
}
```

print a message to help  
identify why the exception  
happened

# Test Exception is Thrown

---

- We can use try-catch in JUnit and test for the string output
- Alternatively, we can test as follows:

```
@Test(expected = IllegalArgumentException.class)
public void testIllegalArgumentExceptionNullCopy() {
    SLList<String> listStr1 = null;
    SLList<String> listStr2 = new SLList<>(listStr1);
}
```

use the optional 'expected'  
attribute of Test annotation

# Checked EmptySLListException

---

- Say we don't allow copy from an empty SLList, and want to enforce it
- We define our own checked exception :

```
public class EmptySLListException extends Exception {  
  
    private String name;  
  
    public EmptySLListException (String name) {  
        super(name);  
        this.name = name;  
    }  
  
    @Override  
    public String getMessage() {  
        return name + " is empty, " + super.getMessage();  
    }  
}
```

inherit Exception functionality

pass the parameter to Exception

override the getMessage()  
method of Exception

call the overridden getMessage()  
method of Exception

# Throws EmptySLListException

---

```
/**
 * Creates a deep copy SLList of other.
 * @param other an SLList object
 * @throws IllegalArgumentException if other is null
 * @throws EmptySLListException if other is empty
 */
public SLList(SLList<T> other) throws EmptySLListException {

    if (other == null) {
        throw new IllegalArgumentException("other is null");
    }

    if (other.size() == 0) {
        throw new EmptySLListException("other");
    }

    sentinel = new Node( i: null, n: null);
    size = 0;

    Node p = sentinel;
    Node q = other.sentinel;
    while (q.next != null) {
        this.addLast(q.next.item);
        q = q.next;
    }
}
```

must advertise

pass the parameter name

# Gotta catch 'em all

---

```
// Try-Catch IllegalArgumentException and EmptySLListException
SLList<String> listStr1 = new SLList<>();
try {
    SLList<String> listStr2 = new SLList<>(listStr1);
} catch (IllegalArgumentException iae) {
    System.out.println("Copy failed, " + iae.getMessage());
} catch (EmptySLListException ese) {
    System.out.println("Cannot copy, " + ese.getMessage());
}
```

must handle the checked exception  
(since we don't throw it)

try by yourself if you set  
EmptySLListException to be unchecked!

# Thank you for your attention !

---

- In this lecture, you have learned to:
  - use or not use null values
  - know the difference between checked and unchecked exceptions in Java
  - use exceptions to signal special results
  - write a copy constructor doing deep copy
- Please continue to Lecture Quiz 6 and Lab 6:
  - to do Lab Exercise 6.1 - 6.3, and
  - to do Exercise 6.1 - 6.3