Exceptions

Exception
Exceptions Overview

• Exceptional Situations
• Syntax of **Try, Catch, Throw**
• Defining and Using Your Own Exceptions
Exceptional Situations

• Exceptions model atypical situations
  – errors in computation or data
  – invalid method parameters
  – failure to complete task
• Exceptions provide the programmer with information regarding problems
  – ex, `NullPointerException` indicates an action on `null` reference
  – some must be handled, others do not (next slide)
• Can handle exceptions by **catching** them
  – respond to exceptions usefully
  – see how to do this in a few slides . . .
• Exceptions allow a method to delegate to caller (method that called the one **throwing** the exception) how the exception is to be handled
Exceptional Situations in Java

- **Exceptions** are classes that extend `Throwable`
  - come in two types:
    - those that must be handled somehow (we'll see how soon), such as `IOException` — e.g., an issue reading a file
    - those that do not; e.g., `NullPointerException`
- **Errors** (far less common, FYI only)
  - could be indirectly caused by your code (such as using up all available memory);
  - could be entirely unrelated
  - you should not attempt to handle these
Exception Handling Syntax (1/2)

• Until now, you have had no control over coping with exceptions. With a `catch` statement, you have the chance to implement your own exception handling

• Process for handling exceptions
  – `try` some code, `catch` exception thrown by tried code, `finally`, “clean up” if necessary
  – `try`, `catch`, and `finally` are reserved words

• `try` denotes code that may throw an exception
  – place questionable code within a `try block`
  – a `try block` must be immediately followed by a `catch block` unlike an `if w/o else`
  – thus, `try-catch` blocks always occurs as pairs

• `catch` exception thrown in `try` block and write special code to handle it
  – catch blocks distinguished by `type` of exception
  – can have several `catch blocks`, each specifying a particular type of exception
  – Once an exception is handled, execution continues after the catch block

• `finally` (optional)
  – special block of code that is executed whether or not an exception is thrown
  – follows `catch block`
Exception Handling Syntax (2/2)

• Here’s the basic syntax:

```java
// Somewhere in your program...
try {
    // Code “in question”
}
catch (most_specific_exception_type name) {
    // Code in response to exception
}
...
finally {
    // Code guaranteed to be executed after try
    // (and previous catches)
}
```

• All parts enclosed in curly braces `{}`
• **try** block comes first
• **catch** block comes after try
  – put exception in parentheses as in method definition
  – you can also have multiple catch blocks
  – formal parameter of type `java.lang.Exception` is the most general and would catch all exceptions (because they are all subclasses)
• **finally** block always comes last
Exception Handling Example 1

• A method call within a `try` block may set off a chain of method calls, the last of which throws an exception
  – Andy tells Wendy to `getADrink();` Wendy tells Sam to `getADrink();` Sam is asleep and throws a `DrinkNotAvailableException` which is defined elsewhere
  – This exception is not a subclass of `RuntimeException`, so it should be caught

```java
public class Andy {
    // Properties, constructor, methods
    // to teach, kayak, eat Chinese food,
    // etc. elided ;)
    public void getWater() {
        try {
            // getADrink() might throw a
            // DrinkNotAvailableException so
            // we have to put it in a try block
            _water = _wendy.getADrink();
        } catch (DrinkNotAvailableException e) {
            this.fire(_wendy);
        }
    }
}
```
Exception Handling Example 2

- **try-catch** blocks can be nested!
  - If Andy’s call to Wendy to `getADrink()` throws `DrinkNotAvailableException`, he can ask Michelle to `getADrink()`.

- **Exception Resolution**
  - similar to method resolution in inheritance hierarchies: starts with the method that throws exception
  - work back up the chain of method calls until a **try-catch** block is found for that exception (or a superclass of that exception)
    - so, you do not necessarily have to **try** and **catch** every exception at every level of calling
    - if an exception must be caught, then you’d better be sure that you catch the exception at some point!
    - if exception is not caught, program will crash or not perform as expected

```java
public class Andy {
    // other code elided
    public void getWater() {
        try {
            _water = _wendy.getADrink();
        } catch (DrinkNotAvailableException e) {
            this.fire(_wendy);
            try {
                _water = _michelle.getADrink();
            } catch (DrinkNotAvailableException e) {
                this.fire(_michelle);
            }
        }
    }
}
```
Defining Your Own Exceptions

• You can define and throw your own specialized exceptions:

```java
throw new DataOutOfBoundsException(…);
throw new QueueEmptyException(…);
```

• Useful for responding to special cases, not covered by pre-defined exceptions
  – you can throw an exception for a different class to catch
  – a method of error handling

• For example:

```java
public class DataOutOfBoundsException extends Exception {
    public DataOutOfBoundsException(String dataName)
    {
        super("Data value " + dataName + " is out of bounds.");
    }
}
```

• The class Exception has a method getMessage(). The String passed to super is printed to the output window for debugging when getMessage() is called by the user
Using Your Own Exceptions (1/2)

- Every method that throws Exceptions that are not subclasses of RuntimeException must declare what exceptions it throws in method declaration.
- `setAge()` is throwing the exception, and we’ll see in the next slide that the exception will be caught in the method that calls `setAge()`.

```java
// Defined in the Person class
public void setAge(int age) throws DataOutOfBoundsException {
    if (age < 0 || age > 120){
        throw new DataOutOfBoundsException(age + "");
    }
    // age+: converts age from int to String
    // Note the constructor takes in a String for message printing
    _age = age;
}
```
Using Your Own Exceptions (2/2)

- Method that calls `setAge()` should have a `try` block surrounding the method call and an accompanying `catch` block to handle `DataOutOfBoundsException`

```java
public void clonePerson() {
    Person clone = new Person();
    int random = (int)(Math.random() * 1000);
    try {
        clone.setAge(random);
    } catch (DataOutOfBoundsException e) {
        System.out.println(e.getMessage()); // gets the message describing the exception that occurred
    }
}
```
QueueEmptyException

- Create your own QueueEmptyException: 
  - Subclass RuntimeException, so that every time dequeue() is called, we don't need to surround it with a try-catch block – same idea behind ArrayIndexOutOfBoundsException
  - If the exception indicates an actual problem, then don't catch it! It should halt the execution of the program

```java
class QueueEmptyException extends RuntimeException {
    public QueueEmptyException() {
        super("Queue is empty");
    }
}
```

```java
class Queue<Type> {
    // QueueEmptyException is a RuntimeException, so we don't need to write "throws" in the method declaration
    public Type dequeue() {
        if (this.isEmpty()) {
            throw new QueueEmptyException();
        }
        // Remaining dequeue() code goes here
    }
}
```
Exceptions: Pros and Cons

• Pros:
  – cleaner code: rather than returning a `boolean` up chain of calls to check for exceptional cases, throw an exception!
  – use return value for meaningful data, not error checking
  – factor out error-checking code into one class, so it can be reused

• Cons:
  – throwing exceptions requires extra computation
  – can become messy if not used sparingly
  – can accidentally cover up serious exceptions, such as `NullPointerException` by catching them
In conclusion

• Words of Wisdom:
  – Never try to “fix up” your program by catching all exceptions
    • “Oh… NullPointerException… let me just catch it, so the TAs won’t know I have buggy code! Hahahaha!!!”
  – Best to throw an exception when an error occurs that you cannot deal with yourself, but can be better handled by some method on the stack

Wow, what an Exception-al lecture!