

Exceptions

Exceptions

- Many problems in the code are handled when the code is compiled, but not all.
- Some are impossible to catch before the program is run.
 - Must run the program to actually determine the values of variables, get user input, etc.
- These errors that occur when the program is running are “exceptions”.

Exception Display

- If an exception isn't handled by the program, it halts the program and prints an error:

```
Exception in thread "main" java.lang.NullPointerException
at SomeClass.method(Someclass.java:54)
at Except.e2(Except.java:12)
at Except.main(Except.java:22)
```

- The error message gives the type of exception that was thrown (NullPointerException)
- ... and the stack trace: list of method calls that led to the exception

Tracing Problems

- The call stack can be very long and contain many methods from the libraries

```
Exception in thread "main" java.util.UnknownFormatConversionException: Conversion = 'i'
at java.util.Formatter$FormatSpecifier.conversion(Formatter.java:2603)
at java.util.Formatter$FormatSpecifier.<init>(Formatter.java:2631)
at java.util.Formatter.parse(Formatter.java:2477)
at java.util.Formatter.format(Formatter.java:2411)
at java.io.PrintStream.format(PrintStream.java:899)
at java.io.PrintStream.printf(PrintStream.java:800)
at Except.e3(Except.java:17)
at Except.main(Except.java:22)
```

- It can be tricky to find the “real” cause.
 - should be the innermost code that isn't fully tested.

The Exception Class

- Exceptions in Java are objects.
- There is a hierarchy of exception types, with the Exception class at the top.
 - You don't generally instantiate Exception; one is automatically created when there is an error.
 - (We will instantiate some exceptions later.)
- But, there are many subclasses of Exception that are used for particular types of errors.

Exception Subclasses

- The subclasses are types that more specifically identify the problem. eg:
 - ArithmeticException: most often caused by a divide-by-zero.
 - IndexOutOfBoundsException: array/List/String index doesn't exist.
 - NoClassDefFoundError: .class file was there when compiled, but it gone now.
- Tend to have informative names: useful errors.

Catching Exceptions

- If an exception occurs, it typically stops the program.
- ... but it can be “caught” or “handled”.
- If code might throw an exception, it can be put in a try block.
 - If it throws an exception, the code in the catch block is executed.

try/catch

- Example:

```
try {
    int x;
    x = 10/0;
} catch(Exception e) {
    System.out.println("an error occurred");
}
```
- If an exception is thrown in the “try”, the “catch” is run.
 - otherwise, it is not.

The Exception Parameter

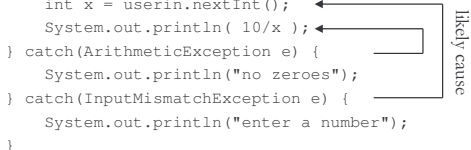
- The catch is a little like an else, and a little like a function definition.

```
try { ... } catch(Exception e) { ... }
```
- The argument (“exception parameter”) is assigned to an appropriate Exception object.
- In the divide-by-zero example, e will be an instance of ArithmeticException.
- The exception parameter’s value can be ignored if you don’t need to use it.

Being Specific

- The exception parameter type allows different types of exceptions to be handled separately:

```
try {
    int x = userin.nextInt();
    System.out.println( 10/x );
} catch(ArithmeticException e) {
    System.out.println("no zeroes");
} catch(InputMismatchException e) {
    System.out.println("enter a number");
}
```



Being Specific

- The exception parameter is matched like arguments to an overloaded function.
 - The type of exception thrown is matched against the exception parameters.
 - That class or any subclass will match.
- eg.
 - Exception matches all exceptions
 - IndexOutOfBoundsException will match both string and array indexing errors

Being Specific

- In general, use a type as specific as possible.
 - Only catch the exceptions you want to catch/can handle properly.
 - Don’t incorrectly handle other problems.
- Other exceptions are propagated.
 - ... and can be caught by the calling function, or farther up the call stack.
 - i.e. the function call might be in a try that catches this exception.

Checked Exceptions

- Unchecked exceptions (`RuntimeException` and children) can be caught or not.
- Checked exceptions (other children of `Exception`) must be handled explicitly.
 - ... so their status can be “checked” by the compiler.
- Checked exceptions must either be handled by a matching `catch` block, or declared as an exception that the method might throw.

Checked Exceptions

```
// Option 1: catch the checked exception
public void method1() {
    try {
        // code that might cause SomeCheckedException
    } catch (SomeCheckedException e) {
        // whatever
    }
}
// Option 2: this method will send the exception up
public void method2() throws SomeCheckedException {
    // code that might cause SomeCheckedException
}
```

The throws Clause

- Used to indicate that the method will “handle” the checked exception by passing it off to the calling code.
 - Similarly, the calling code must either catch it or be declared indicating that it throws the exception.

```
public int myMethod(...) throws
    Exception1, Exception2 { ... }
```
- Any uncaught checked exceptions must be listed in the `throws` clause.

Throwing Exceptions

- If your code gets to a situation it can't handle, it can throw an exception.
 - eg. constructor/setter receives illegal value
- Create an instance of the appropriate exception class:

```
new SomeException("Error message");
```
- Throw the exception with a “`throw`” statement.

```
throw new SomeException("no negatives");
```

Example

- From the `Student` class:

```
public void setFirstName(String name) {
    if ( name.length() > 0 ) {
        firstName = name;
    } else {
        throw new IllegalArgumentException
            ("Name must have length >0");
    }
}
```