

Advanced Programming

CSE 201

Instructor: Sambuddho

(Semester: Monsoon 2024)

Week 4 – Inner Classes
and Exceptions/Exception
Handling

Inner Classes

```
public class TalkingClock
{
    private int interval;
    private boolean beep;

    public TalkingClock(int interval, boolean beep) { . . . }
    public void start() { . . . }

    public class TimePrinter implements ActionListener
    {
        // an inner class
        . . .
    }
}
```

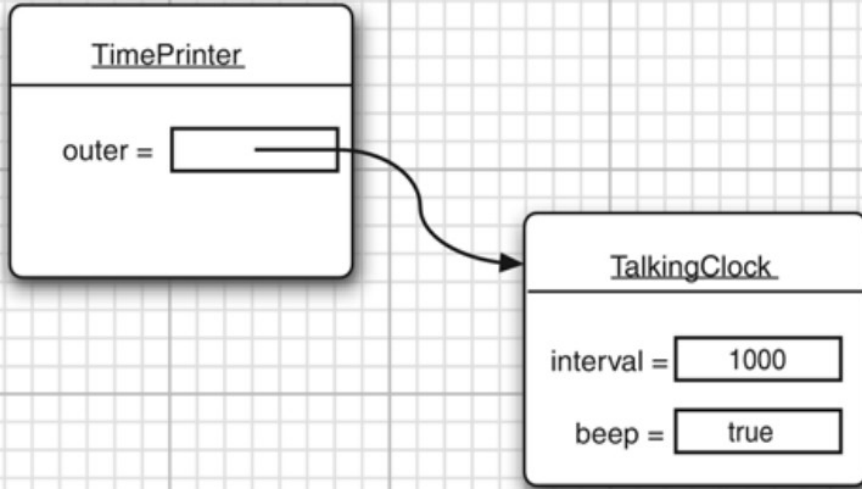
Inner

Classes

```
public class TimePrinter implements ActionListener
{
    public void actionPerformed(ActionEvent event)
    {
        System.out.println("At the tone, the time is "
            + Instant.ofEpochMilli(event.getWhen()));
        if (beep) Toolkit.getDefaultToolkit().beep();
    }
}
```

Implicit reference to object of outer class. However the correct way to use is to refer with object of the outer class.

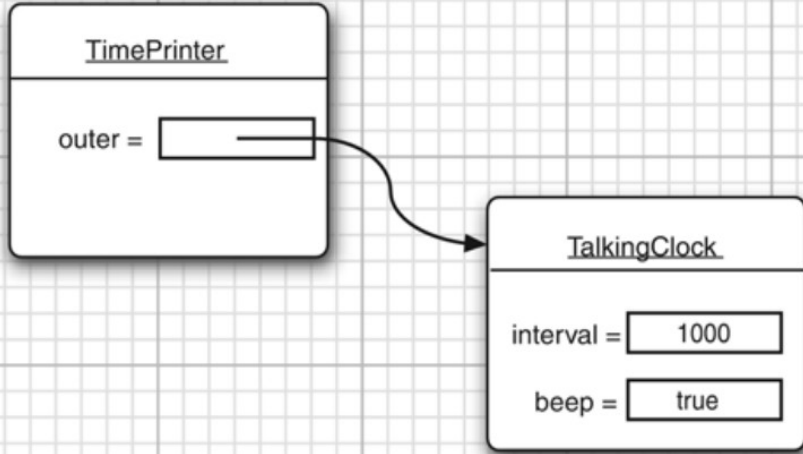
Inner Classes



```
public TimePrinter(TalkingClock clock) // automatically generated code
{
    outer = clock;
}
```

Accessible with an
object of the outer
class in the inner.

Inner Classes



```
public void actionPerformed(ActionEvent event)
{
    . . .
    if (TalkingClock.this.beep) Toolkit.getDefaultToolkit().beep();
}
```

Accessible without
requiring in object.
Outer class
encapsulates the inner
class as well.

Anonymous Class

```
new SuperType(construction parameters)  
{  
    inner class methods and data  
}
```

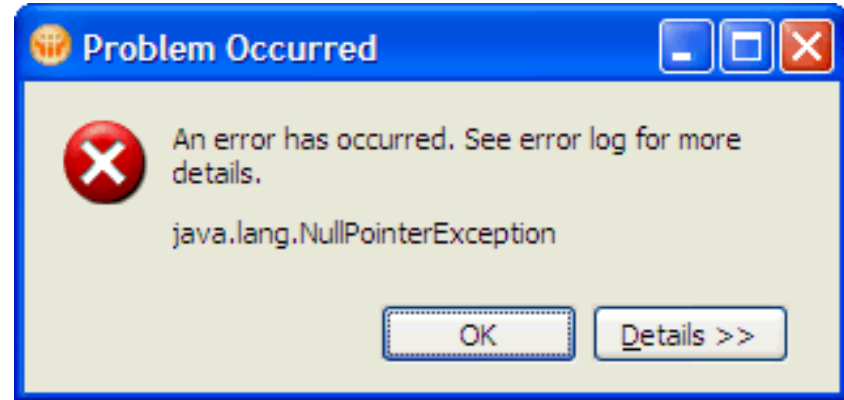
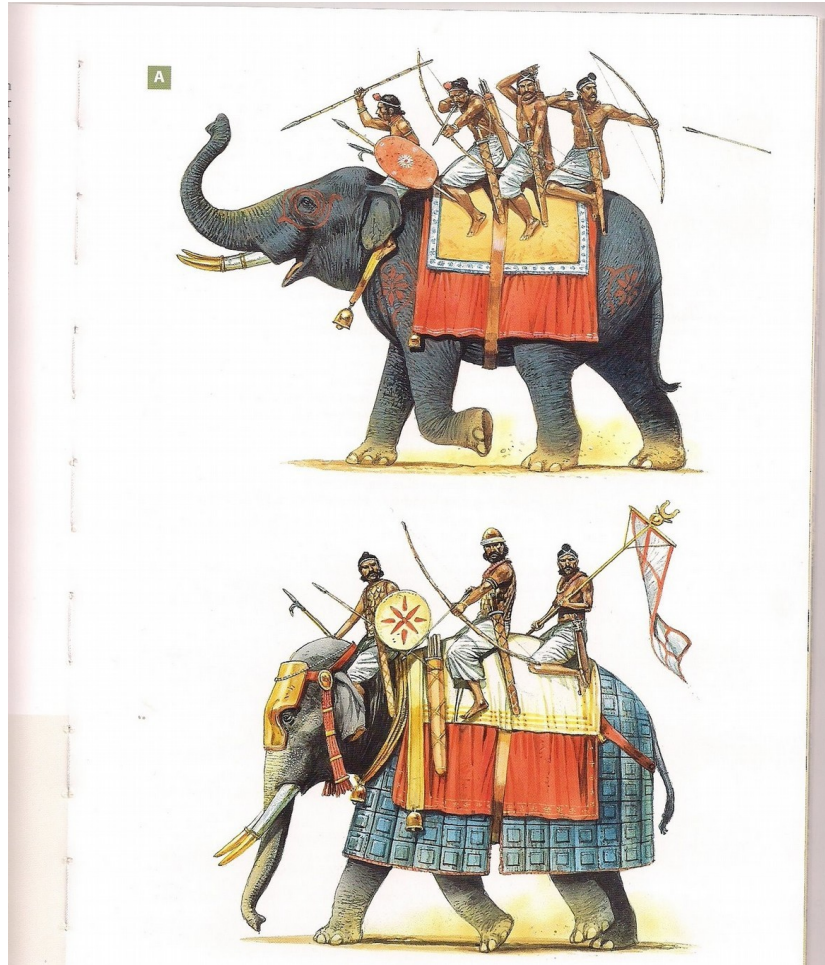
```
public void start(int interval, boolean beep)  
{  
    var listener = new ActionListener()  
    {  
        public void actionPerformed(ActionEvent event)  
        {  
            System.out.println("At the tone, the time is "  
                + Instant.ofEpochMilli(event.getWhen()));  
            if (beep) Toolkit.getDefaultToolkit().beep();  
        }  
    };  
    var timer = new Timer(interval, listener);  
    timer.start();  
}
```

Exceptions and Exception Handling

OOP avatar of errors and error handling.

- Array out of bounds.
- IOException – e.g. File not found error.
- Mathematical exception – divide by zero error, NAN error, floating point exception *etc.*

Being Defensive is Important

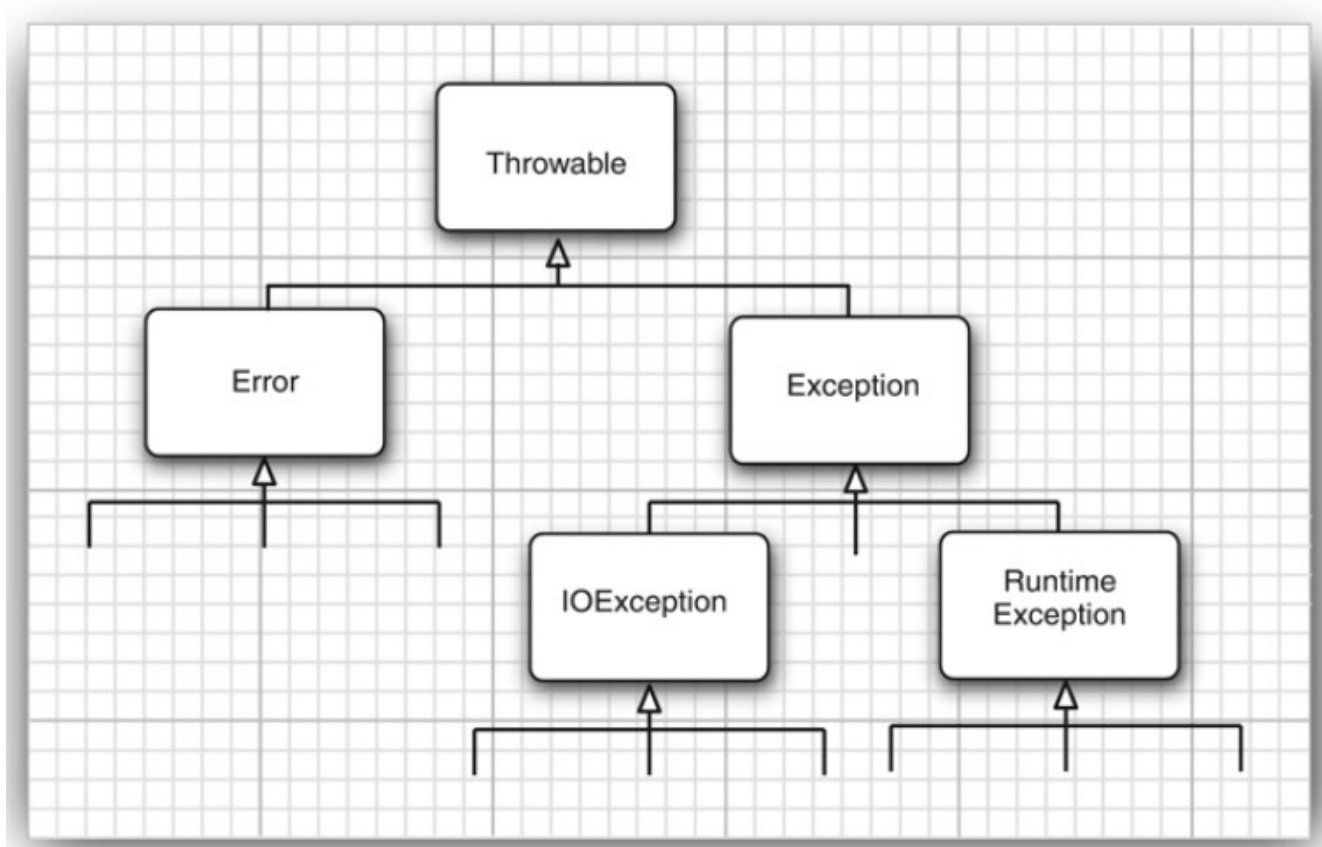


www.shutterstock.com · 109665452

Exception Handling Syntax

- 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*

Exceptions and Exception Handling



Exceptions
are *also*
classes

Exceptions Handling by JVM

- Any method invocation is represented as a “**stack frame**” on the Java “**stack**”
- **Callee-Caller** relationship: If method A calls method B then A is **caller** and B is **callee**
- Each frame stores local variables, input parameters, return values and intermediate calculations
 - In addition, each frame also stores an “**exception table**”
 - This exception table stores information on each try/catch/finally block, i.e. the instruction offset where the catch/finally blocks are defined.

How JVM handles exceptions:

1. Look for exception handler in current stack frame (method)
2. If not found, then terminate the execution of current method and go to the callee method and repeat step 1 by looking into callee's exception table
3. If no matching handler is found in any stack frame, then JVM finally terminates by throwing the stack trace (printStackTrace method)

Exception Handling Syntax

- 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*

Exceptions and Exception Handling

Methods tells Java compiler that what kind of errors it can throw.

```
class MyAnimation
{
    . . .
    public Image loadImage(String s) throws FileNotFoundException, EOFException
    {
        . . .
    }
}
```

Throwing an Exception

```
String readData(Scanner in) throws EOFException
{
    . . .
    while (. . .)
    {
        if (!in.hasNext()) // EOF encountered
        {
            if (n < len)
                throw new EOFException();
        }
        . . .
    }
    return s;
}
```

Creating and Throwing Exception Class

```
class FileFormatException extends IOException
{
    public FileFormatException() {}
    public FileFormatException(String gripe)
    {
        super(gripe);
    }
}
```

```
String readData(Scanner in) throws FileFormatException
{
    . . .
    while (. . .)
    {
        if (ch == -1) // EOF encountered
        {
            if (n < len)
                throw new FileFormatException();
        }
        . . .
    }
    return s;
}
```

Catching What Was Thrown

```
try
{
    code
    more code
    more code
}
catch (ExceptionType e)
{
    handler for this type
}
```

In try {} code after the throw is skipped.

The program jumps to the catch() handler.

If no appropriate handler, then JRE handles it and show it on stdout

Catching What Was Thrown

```
try
{
    code that might throw exceptions
}
catch (FileNotFoundException e)
{
    emergency action for missing files
}
catch (UnknownHostException e)
{
    emergency action for unknown hosts
}
catch (IOException e)
{
    emergency action for all other I/O problems
}
```

```
try
{
    code that might throw exceptions
}
catch (FileNotFoundException | UnknownHostException e)
{
    emergency action for missing files and unknown hosts
}
catch (IOException e)
{
    emergency action for all other I/O problems
}
```

Combining exceptions

Rethrowing Exceptions

```
try
{
    access the database
}
catch (SQLException e)
{
    throw new ServletException("database error: " + e.getMessage());
}
```

Finally Clause

- Executed at the end of all `try{} catch{} blocks`.
- Last set of instructions to be called before the program terminates. Usually used for resource release and cleanup operations.

Finally Clause

```
var in = new FileInputStream(. . .);  
try  
{  
    // 1  
    code that might throw exceptions  
  
    // 2  
}  
catch (IOException e)  
{  
    // 3  
    show error message  
    // 4  
}  
finally  
{  
    // 5  
    in.close();  
}  
// 6
```

Using Assertions

- Idiomatic tools for defensive programming.
- Faster execution than throwing exceptions.
- Not to be used for everyday programs.
- Irrecoverable.
- Usually program terminates.

```
assert <condition>;
```

```
assert <condition> : <expression> ;
```

[Check condition. If false then create an object with argument <expression> of type AssertionError – JVM catches it and prints the details presented in the <expression>]

Using Assertions

```
if (x > 0){  
    ....  
}  
Else {  
    ...  
}
```

```
if (x > 0){  
    throw new  
exception("myexception")  
;  
}
```

```
catch(Exception ep){  
    ...  
}
```

```
assert x>0 : new  
String("x>0");
```

Logging

- `System.out.println()` cannot be always used – makes things slow.
- Adding and removing them at all places can be cumbersome.
- Usually not used in production code.
- Logging can be collectively enabled or disabled.

```
Logger.getGlobal().info("File->Open menu item selected");
```

```
May 10, 2013 10:12:15 PM LoggingImageViewer fileOpen  
INFO: File->Open menu item selected
```

Log Levels

SEVERE

WARNING

INFO

CONFIG

FINE

FINER

FINEST

Top -> Bottom levels of logging.

If you log a bottom level, then you log all levels above it.

SEVERE, WARNING and INFO are always enable for every Java program by default.

```
logger.warning(msg);  
logger.fine(msg);
```

```
logger.log(Level.<Level  
name>, msg);
```


Logging Unexpected Exceptions

Two methods commonly used:

```
void throwing(String className, String methodName, Throwable t)
```

```
void log(Level l, String message, Throwable t)
```

```
    try
    {
        if (. . .)
        {
            var e = new IOException(". . .");
            logger.throwing("com.mycompany.mylib.Reader", "read", e);
            throw e;
        }
    }

    catch (IOException e)
    {
        Logger.getLogger("com.mycompany.myapp").log(Level.WARNING, "Reading image", e);
    }
```