Building Enterprise Database Applications

using Rational Rose, Java and Mr Architecture

A short course by Kade Hansson

Course Contents

- **UML, object-orientation and design patterns**
- **Java language and essential APIs**
- **Java GUI components and event model**
- **Java I/O and TCP/IP sockets**
- **JDBC, Servlets and JSPs**
- **Mr Architecture**

What is Java?

- What is Java?
  - A programming language invented by Sun Microsystems
    - **Strongly-typed**
    - Object-oriented (**class-based**)
    - **Imperative** (i.e. not "functional" or "logical")
    - Syntax closely matches C
  - A "write once, run anywhere" runtime system
    - A **virtual machine** employing an **interpreter** and/or a **JIT compiler**
    - A standard set of **APIs** (application program interfaces)

The Building Blocks of Java

- Java programs are built from classes
- A class is a **type definition** for an object, and also an "object" in itself– **Singleton** pattern
  - Members belonging to the class are in **class scope**
  - Members belonging to objects (instances of classes) are in **instance scope**
- All classes (except *Object* itself) are **subclasses** of the Java-defined class *Object*
- Each class (except *Object*) has a single **direct superclass** it may inherit from i.e. Java does not support multiple inheritance
  - ⇒ each class may have many **superclasses** it may inherit from, but they form a unique trace back to the primordial class *Object*

Interfaces

- Java does not support multiple inheritance, but it does support a second definition-only inheritance hierarchy which allows a form of it
- A special kind of class, called an **interface**, defines only abstract methods and {frozen} fields (UML defines a similar construct without allowing for constants)
- An interface may have zero or many **direct superinterfaces**, and contains the union set over all methods and fields defined in them
- A class may also have zero or many **direct inplemented interfaces**, but if it is not abstract itself, it must define implementations for all methods in the union set over all methods in all (both direct and indirect) **implemented interfaces**
- Because interfaces are abstract by definition, it is redundant (in both UML and Java) to include any indication of this (similarly for interface methods)

- *In most situations, when we talk about classes, subject to the above restrictions, we include interfaces*

How Does Java Work?

- Programs are written as text files with **.java** extension
  - One public class per file (exception for **inner classes**)
  - If a public class appears:
    - it must have the same leaf name as the file
    - it should appear in a directory structure matching the package name (i.e. *au.gov.tas.dpiwe.mr.Container* should appear in *au/gov/tas/dpiwe/mr*)
- Programs are created by compiling **.java** files into **.class** files
  - One **.java** file may produce many **.class** files
- Programs are run by interpreting or compiling **.class** files
  - Even when compilation is used, it is rare that object binaries are produced (this would negate the benefits of write once, run anywhere)

Java Type System

- Divided into two parts:
  - ***primitive*** **(`boolean`, `byte`, `short`, `int`, `long`, `float`, `double`, `char`) and** ***reference*** **(to objects or arrays)**
- Java is strongly-typed
  - A primitive value of a particular type cannot be used directly in a context where another primitive type is required, unless that type is convertible to the required type by an automatic **widening conversion**
  - A reference of a particular type can only be used in a context where a reference to that type, a superclass type or an implemented interface is required
  - A primitive value (of any type) cannot be used where a reference type is required and a reference of a particular type cannot be used where a primitive is required
  - A **narrowing conversion**, where a reference to one type is converted to a reference to a subclass or subinterface, or where a primitive type is reduced in precision or width, can be achieved by a cast expression: (*Type*)*Expression*

What is in a Java Source File?

- One package directive (or none, implying the default package)
  - Package names are typically reversed internet domain names followed by further organisation defined naming conventions
    (e.g. `package au.gov.tas.dpiwe.mr;`)
- Multiple import directives (or none– although `import java.lang.*;` is implicit)
  - Exception is made for Java language packages, Java extension packages and some vendors
    (e.g.
    ```
    import java.util.*;
    import oracle.jdbc.driver.OracleResultSet;
    ```
)
- Multiple class or interface declarations
  (e.g.
  ```
  public abstract class Container extends BeanBundler
          implements UserTransaction, Principal { ... }
  ```
)
- No include directives or other compiler preprocessor directives

Class Heading

- A class declaration consists of multiple modifiers followed by the reserved keyword **`class`**
  - Legal class modifiers include:
    - Abstraction modifers — **`abstract`** or **`final`**
    - Visibility modifiers (slightly different from UML) — **`public`**, **`protected`**, none (default) or **`private`**
    - Class scope inner class modifier — **`static`**
- A class continues with the **class leaf name**, the keyword **`extends`** followed by the (possibly fully-qualified) **direct superclass name** (optional), then the keyword **`implements`** followed by a comma separated list of (possibly fully-qualified) **implemented interface names** (optional.) The body follows this heading.

Interface Class

- A interface class declaration constists of a visibility modifier followed by the reserved keyword **`interface`**
  - Visibility modifiers (again, different from UML) — **`public`**, **`protected`**, none (default) or **`private`**
- An interface continues with the **interface leaf name**, then the keyword **`extends`** followed by a comma separated list of (possibly fully-qualified) **direct superinterface names** (optional.) The interface class body follows.
- An interface class body contains only abstract methods, which have no bodies
  - The modifier **`abstract`** is not used on the method, just as it is not used on the interface itself
  - Instead of a body, an interface method declaration ends with a semi-colon `;`
    e.g. `public int compareTo(Object other);`
    (similarly for **`abstract`** methods in **`abstract`** classes)

Class Body

- Classes are bracketed using the block bracketing convention from C:
  - A class body begins with an open curly brace `{`
  - A class body ends with a close curly brace `}`
  - The class body is typically indented from the brackets by a tab or two or three spaces (Java convention)
- A class body consists of **member declarations** and **initializers**
  - A **member** is one of:
    - A **field** (also called an attribute in UML)
    - A **method** (also called an operation in UML)
    - An **inner class**, which follows the same syntax as an outer class

Field Declarations

- A field declaration consists of (followed by a semi-colon ; delimiter):
  - visibility modifiers (slightly different meaning from UML) — **`public`**, **`protected`**, none (default) or **`private`**
  - volatility modifier (to resolve multi-threading issues) — **`volatile`**
  - persistence modifier — **`transient`**
  - {frozen} modifier — **`final`**
  - class scope modifier — **`static`**
  - type (possibly fully-qualified) — e.g. **`int`** or *Vector*
  - name — e.g. *value*
  - initializer (optional, preceded by equals =) — e.g. **`new int`**`[] {1}`

Method Heading

- A method heading consists of (followed by a method body or semi-colon ;):
  - visibility modifiers (slightly different meaning from UML) — **`public`**, **`protected`**, none (default) or **`private`**
  - concurrency lock modifier (to resolve multi-threading issues) — **`synchronized`**
  - floating point modifier — **`strictfp`**
  - abstraction/implementation modifier — **`abstract`** or **`native`**
  - class scope modifier — **`static`**
  - return type (omitted for **constructor**) — e.g. **`void`** or *String*
  - name, parameter types and formal names — e.g. *getValue()*
  - checked exception clause — e.g. **`throws`** `A, B`

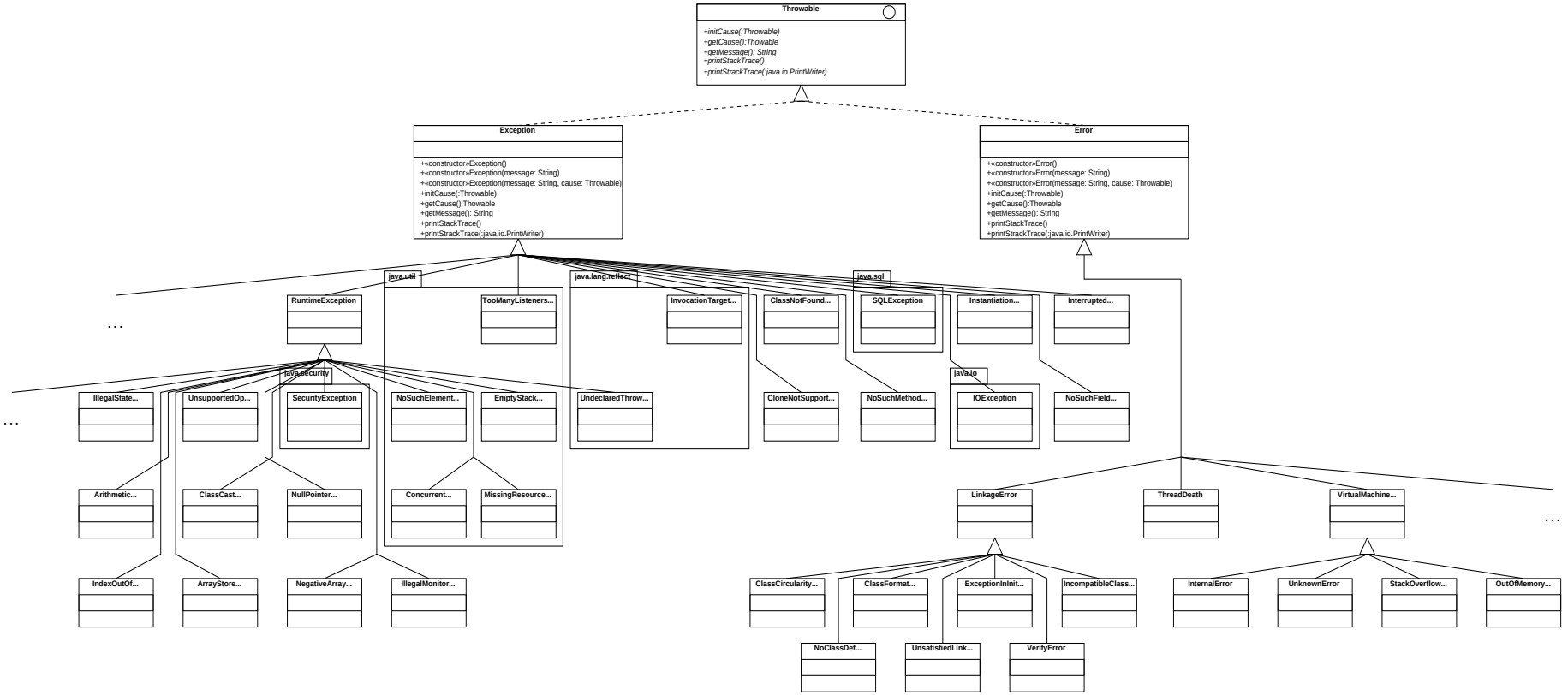
Method Signature

- A method's signature is the combination of a method's:
  - name
  - formal parameter types
  - return type
- In Java, one can only... **overload** when... **override** when...
  - the formal parameter types *are*... *different* (and return type *is*) *identical*
- In a method heading, formal parameters are listed in a comma-separated list delimited by parentheses `( )`
  e.g.
```
public void operation(ParameterTypeA parameterA,
                      ParameterTypeB parameterB)
```
- All primitive type (**`int`**, **`float`**, **`char`** etc.) parameters are "in" (UML) **values** and all reference types are "in" (UML) **references** (or pointers in C nomenclature)

Checked Exceptions

- A **checked exception** is an exception which a method declares itself to generate (or **throw**) during unusual situations particular to that method
- An **unchecked exception** is an exception which any method might throw due to an unexpected or unhandled condition occuring (e.g. *NullPointerException*)
- Unchecked exceptions are subclasses of the Java type *RuntimeException*
  - **All other exceptions are *checked exceptions.* Any method which may throw a checked exception, either directly using a `throw` statement, or indirectly by calling a method which declares a checked exception which is not subsequently *caught* by the calling method, must declare that exception in the `throws` clause of its heading.**
- An **exception** is an unusual situation encountered during processing
- Exceptions are represented by subclasses of the Java type *Exception*
- An **error** is an unexpected condition in the runtime environment
- Errors are represented by subclasses of the Java type *Error*
- *Errors* and *Exceptions* are instances of the Java type *Throwable*

The Throwable Hierarchy



Instance Scope versus Class Scope

- The body of an **instance scope** method or an initializer may refer to:
  - members of the class or instance as if they were local variables or parameters, unless an identical name is used for a local variable or parameter
  - members of the instance using the explicit notation `this`.*MemberName*
  - class scope members (and class scope members in other classes, where visibility permits) using the notation *ClassName*.*MemberName*
- The body of a **class scope** method or initializer may refer to:
  - members of the class only as if they were local variables or parameters, unless an identical name is used for a local variable or parameter
  - class scope members (and class scope members in other classes, where visibility permits) using the notation *ClassName*.*MemberName*

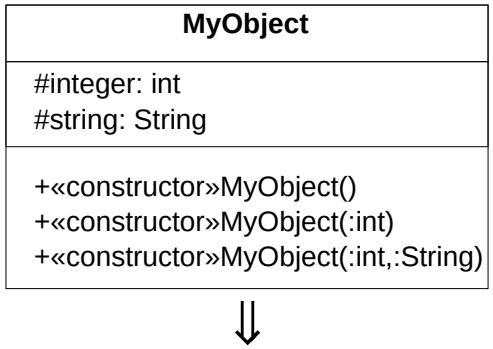
- N.B. here, the *MemberName* of a method must include an actual parameter list

Superclass Scope and Constructor Delegation

- The body of an instance scope method or an initializer may refer to:
  - members of the superinstance using the explicit notation **`super`**`.`*MemberName*, particularly in the case where an identical name is used for a local variable, parameter or subclass member
  - in a constructor only, an immediate superclass constructor, but only as the first statement, using the notation **`super`**`(`*ActualParameterList*`)`
  - in a constructor only, another constructor in the same class, but only as the first statement, using the notation **`this`**`(`*ActualParameterList*`)`

- N.B. here again, the *MemberName* of a method must include an actual parameter list

«constructor» Methods



⇓

```
class MyObject {
    int integer;
    String string;
    public MyObject() {
        integer=null; string=null;
    }
    public MyObject(int integer) {
        this(integer,null); // Special notation to
                            // call constructor below
    }
    public MyObject(int integer, String string) {
        this.integer=integer; this.string=string;
    }
}
```

Initializers and Class Scope

- A class scope member or initializer is prefixed by the keyword **`static`**, exists without reference to any instance, and is effectively shared among any instances that do exist. For example...
  - ```
    class CountedObject {
       private static int instanceCount=0;
       public CountedObject() { instanceCount++; }
       public int getInstanceCount() { return instanceCount; }
    }
    ```
  - ```
    class SharedResourceManager {
       private static SharedResource sharedResource=new SharedResource();
       static {
           sharedResource.init();
       }
       public static SharedResource getSharedResource() {
           return sharedResource;
       }
       public void finalize() { sharedResource.destroy(); }
    }
    ```
- Instance initializers outside field declarations are much less common– they are executed prior to the constructor but without knowledge of constructor parameters

Inner Classes and Class Scope Inner Classes

- Inner classes may appear in class bodies like any other class members
- Each inner class instance (except one which has class scope) is associated with its parent instance(s)
  - Inner classes can refer to any member of its parent instance(s) as if it were a member of the inner class unless there is an identically named member in the inner class or its superclasses
  - Inner classes can refer to any parent instance explicitly using the notation *ClassName*`.this` (and to members using the notation *ClassName*`.this.`*MemberName*)
- Class scope inner classes cannot refer to parent instances at all, but they can refer to static members using implicit or explicit notation

Method and Initializer Body

- The body of a method (constructor or class scope) or initializer (class or instance scope) is called a **block**
- Blocks nest, and consist of **statements** or sub-blocks– **Component** pattern
- A block is delimited by curly braces `{ }`, and is indented as for a class body
- Statements appearing in a block are separated by semi-colons `;`
- The last statement in a block must be followed by a semi-colon `;`
- Some statements may contain blocks as part of their syntax
- Other statements must be separated from anonymous sub-blocks which follow them by semi-colons
  - N.B. `if (flag); { System.out.println("flag true"); }`
    (displays *flag true* always) has different semantics to
    `if (flag) { System.out.println("flag true"); }`
    (displays *flag true* only if `flag==true`)

Java Statements

- Empty statement
- Declaration (statement)
- Expression statements
- Conditional and selection statements

```
;

int value=1;

a=b+1;
a();
new Hashtable(10);

if (flag) a(); else b();
if (condition1) {
  a();
} elsif (condition2) {
  b();
}
switch (primitive) {
  case 1: a(); break;
  otherwise: b();
}
```

Java Statements (continued)

- Iteration statements

```
for (int i=0; i<10; i++)
  a(i);
while (loop) a();
do {
  a(); b();
} while (loop);
```

- Exception handler

```
try {
  a();
} catch (IOException x) {
  b();
} catch (Exception x) {
} finally {
  c();
}
```

- Loop or method escape statement

```
break; or return;
```

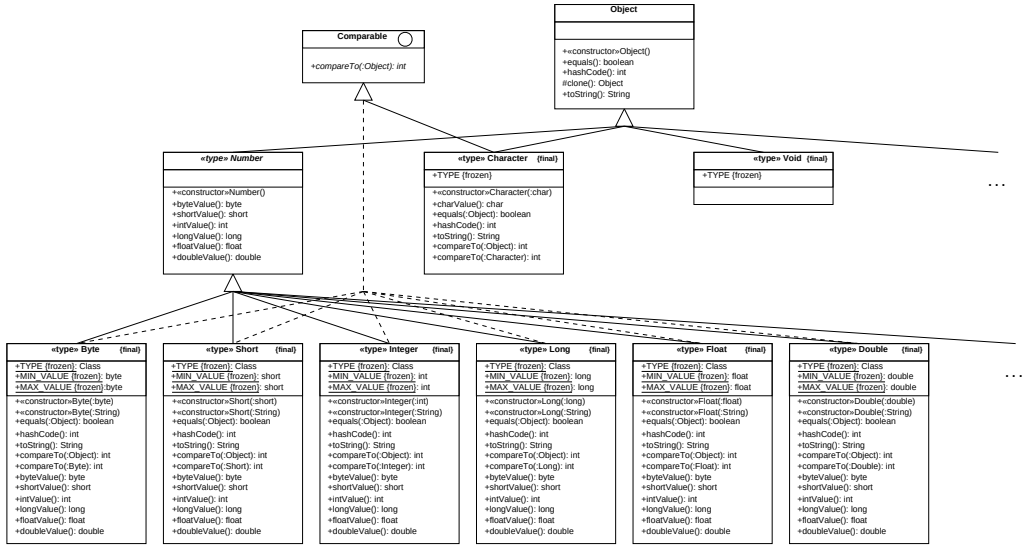
Java Operators

- Assignment operator `=`
- Arithmetic operators
  - add, subtract, multiply, divide, modulus `+, -, *, /, %`
  - add, subtract, multiply, divide, modulus and accumulate `+=, -=, *=, /=, %=`
  - pre- or post-increment, positive or negative (unary) `++, --, +, -`
- Bitwise and logical operators
  - or, and, exclusive-or `|, &, ^`
  - or, and, exclusive-or and accumulate `|=, &=, ^=`
  - short-circuit or, and `||, &&`
  - bitwise not, logical not (unary) `~, !`

Java Operators (continued)

- Comparison operators
  - less, less-equal, greater, greater-equal, equal, not-equal `<`, `<=`, `>`, `>=`, `==`, `!=`
- Shift operators
  - shift left, shift right, rotate right `<<`, `>>`, `>>>`
  - shift left, shift right, rotate right and accumulate `<<=`, `>>=`, `>>>=`
- Conditional operator
  - The conditional operator provides for selection of one of two possible computations based on the result of a boolean computation
    e.g. *flag*?*whenTrue*:*whenFalse*
- String concatenation operator `+`
  (if just one operand is String, do *toString()* on object wrapper of other)

Object Wrappers for Primitive Types



- Each primitive type has its own object wrapper
- Object wrappers are **immutable**
  - *Once an object wrapper is constructed, its value cannot change*
  - Immutability leads to the application of the class stereotype **«type»** in UML

Java Literals

• Null reference literal	`null`
• Boolean literals	`true`, `false`
• Numeric literals	
• Octal (`int`)	`-0173`
• Octal (`long`)	`-0173L`
• Decimal (`int`)	`-123`
• Decimal (`long`)	`-123L`
• Decimal with fraction	`-123.456`
• Exponential decimal (-123.456×10^{-78})	`-123.456E-78`
• Hexadecimal	`-0x7B`

Java Literals (continued)

- Character literals
  - Non-escaped `'a','1','&'`
  - Escaped `''','\'','\\'`
  - Special characters `'\r','\t','\n'`
  - Unicode characters `'\u2297','⊗'`
- String literals
  - Non-escaped `"Hello world!"`
  - Escaped `"My name is \"Fred\""`
  - Special characters `"Line 1\nLine 2"`
  - Unicode characters `"Unicode 2297 is ⊗"`

Essential Java APIs

- The *System* class
  - ***System.in***
  - ***System.out***
- The *PrintWriter* class
- The *Array* and *Arrays* classes
- The *Math* class
- The *Collection* API
  - **Interfaces**
    - ***Collection, List, Set, SortedSet, Map, SortedMap, Iterator***
  - **Abstract classes**
    - ***AbstractCollection, AbstractList, AbstractSet, AbstractSortedSet, AbstractMap, AbstractSortedMap***
  - **Implementations**
    - ***Vector, LinkedList, HashSet, TreeSet, HashMap, TreeMap***