

# Java 105/200

## 3 - Advanced object oriented programming





Designing a standard access to multiple classes

# INTERFACES



# Definition

- An interface is a set of abstract method signatures and constants
- An interface is a data type that acts like a superclass.
- Use an interface when you've already used inheritance.
  - Java does not support multiple inheritance, the ability to inherit from two or more classes.



# Internal vs. external

- An internal interface (building on)
  - Combines shared behavior into a datatype
  - Is implemented by Java's interface type
- An external interface (calling to)
  - Are the public methods exposed by a component
  - Are published as the javadocs of the component
  - Is said to be the API of the component
- GUI type is for visual mapping of user intent



# Typical discovery

- Interfaces are found during analysis often when two or more classes exhibit the same behavior
- A common name is defined
  - I<noun> or I<verb>
- The interface is created
- The classes declare the interface
- The interface is used as a reference type



# Creating an interface

- The name of the interface describes an ability to do something
  - Clone**able**, Compar**able**, Serializ**able**, Runn**able**
  - IProcess, IMove, IMessage, ISortReverse
- The class-like declaration is
  - `public interface <name> { }`



# Interfaces as markers

- Approval only, no method implementation required
- Common interfaces
  - Serializable
  - Cloneable
  - EventListener
- Better as an annotation for new markers
  - Needs backwards compatibility for old markers



# Writing interface methods

- Interface methods are always by default
  - public abstract
- Interface fields are always by default
  - public static final
- End methods with ; and not a code block.
  - Object clone( );
  - void read( );





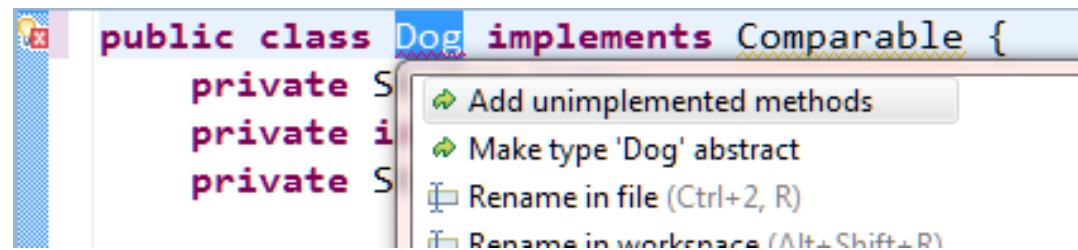
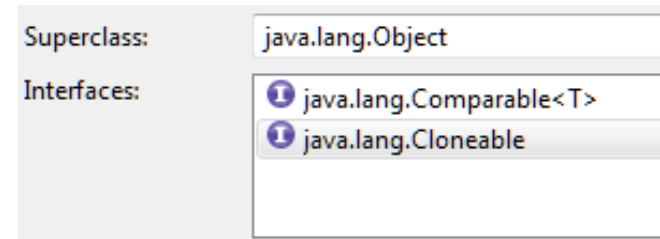
# Implementing interfaces

- Interfaces can be used with or without superclasses
- Interfaces use the keyword **implements**
  - `public class Bird implements IFly { }`
- Classes can implement multiple interfaces
  - `public class Bird implements IFly, ICommunicate, INest { }`



# IDEs and interfaces

- Can ask for interfaces on class creation
- Adding an interface after class creation creates a light bulb which will implement methods.





# Exercise (200)

- Write an interface `IMotorized` with methods
  - `startMotor( )`, `stopMotor( )`
- Add classes and implement interfaces
  - `GoCart`, `Canoe`, `Speedboat`, `Lawnmower`, `RidingLawnmower`, `ReelMower`
  - Optionally, other methods like `mow( )`
- Write other interfaces and apply
  - `IControlMotion`, `IControlDirection`, `ISubmerge`, `ITransport`
- Use inheritance where useful
- Start motors, move them all, stop moving, stop motors



# Static imports (1.5)

- Static imports make code more readable.
- Constants usually defined in interfaces
- Imports all constants without need to preface with class name



# Static imports (1.5)

- before
  - `import java.awt.Color;`
  - `class Employee {`
  - `... Color.BLUE ... }`
- after
  - `import static java.awt.Color.*;`
  - `class Employee {`
  - `... BLUE ... }`



# Interface default methods (1.8)

- Used for anonymous objects
  - Not necessary to override
  - a call from overridden method is `<classname>.super.printStuff();`
- `@Override`
- **default void printStuff() {**
  - `System.out.println("interface default method");`
- **}**



Packages, import, CLASSPATH

# CLASS ORGANIZATION



# Packages

- A package logically organizes classes into groups
  - A Dog class in two different packages will not be confused
    - monday.Dog
    - tuesday.Dog
- A package physically organizes class files into a directory





# Declaring a package

- Putting a class in a package requires one line at the top of the file before the class declaration.
  - **package** monday;
  - **public class** Dog { }



# Nested packages

- Packages can be nested when convenient
  - `monday.exercises;`
  - `java.lang`
  - `java.util`
  - `java.util.zip`
- `javax` packages refer to classes from 1.2 on
  - `javax.net`
  - `javax.print`
  - `javax.sql`



# Qualified class names

- A class is fully named with its package name
  - monday.Dog
  - java.lang.String
  - java.util.zip.ZipFile



# Troubleshooting steps

scope	Limiting factor
Java tool in OS →	PATH
java →	PATH
package.class in OS → package.class in IDE → package.class in java → class	CLASSPATH reference in IDE <code>import</code> (optional) access modifier
method	access modifier

# Providing access to package classes



- You must first provide physical access (project reference) to a class you want to use
  - java.lang classes are available by default
- Use the keyword **import** above the class declaration
  - `import java.util.Date;`
  - `import java.math.BigDecimal;`

# Providing access to package classes



- Provide access to all package classes with `*`
  - `import java.math.*;`
  - This is a search help only, it will not physically copy the code so there is no inefficiency.
  - `BigDecimal price = new BigDecimal("39.95");`
- Provide access to a package class without import by using qualified name.
  - `java.math.BigDecimal price = new java.math.BigDecimal("39.95");`



# Package access keywords

- `<none>` - restricts access to package only
- **protected** – same as `<none>` but allows access by subclasses
  - Used for access into a library only after extending the code
- Access modifiers must be compatible with overriding methods.



# Exercise - protected

- Change a get method in Person from public to
  - Package access
  - Private
  - Protected
- And see if you can access it from your package





# Exercise – using external jars

- Go to [apache.org](http://apache.org) and get a jar file from the Commons library
  - Projects > Language > Java
  - Choose a project like
    - Apache Commons Lang
    - Apache Commons Math
    - Apache Commons Net
  - Choose download site (and mirror)
  - Download binaries and source (and docs if separate).



# Eclipse external jars

- Project Properties > Java Build Path > Libraries > Add External JARs...
  - Choose location of JAR
- Add source code
  - Select JAR file > click outline arrow on left
  - Select Source attachment > Edit...
  - Choose External File.. Location
- Add Javadocs if available.



# Eclipse external jars

- Confirm under project > Referenced Libraries
- Easy method:
  - Drag and drop file from Explorer window to project folder.



# Exercise – external jars

- Import package
- Use classes



Try-catch, throw, throws, finally

# EXCEPTION HANDLING



# Old school

- try some code
- if (rule is broken) {
  - if( first type of error ) handle this error
  - else if (second type of error) handle it
  - ...
  - else do the default error handling thing
- }
- try some more code
- if (...



# OO school

- write error handling code **FIRST**
- try {
  - code
  - code
  - code
- }
- the JVM routes you to a kind of **case** structure you write



# OO error handling

- Benefits
  - reusable error handling
  - customized error types
  - simple structure creates easy to read code
- Not benefits
  - writing the same amount of code
  - nesting structures creates hard to read code





# try and catch

- **try** {
  - code that can fail
- }
- **catch** (ExceptionDatatype objectCaught) {
  - do something about it
- }



# Catching what?

- Exception type objects are managed by JVM
- Objects are created by JVM when
  - code fails to pass explicit tests
  - code fails for other reasons
- Objects are routed to the **catch** block by the JVM
- Code is run by matching object to catch parameter type just like a method



# Two exception types

- Compile time (checked)
  - Running code that can create an exception **must** be in a **try** block
  - Code will not compile otherwise
- Runtime (unchecked)
  - Exceptions can be caught and handled if wanted
  - Exceptions not caught cause program failure.



# Unchecked exceptions - IDEs

- You can always wrap code in a try/catch
- Surround with try-catch

```
byte[] emptyArray = new byte[0];  
System.out.println(emptyArray[0]);
```





# Unchecked exceptions

- Remove comment, print the message
- Always initialize variables outside of try/catch so they are in scope in or out of those blocks.

```
byte[] emptyArray = new byte[0];  
try {  
    System.out.println(emptyArray[0]);  
} catch (Exception e) {  
    System.out.println(e);  
}
```



# Exercise

- Create an array and access an element that is not in range. Add a println statement before and after access.
- Wrap the access statement in a try/catch and run again.
- Try dividing by zero.
- Wrap in a try/catch and print out the exception object.



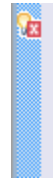
# Checked exceptions - IDEs

- Checked exceptions will show a light bulb error.



```
System.out.println("before sleeping");  
Thread.sleep(4321);  
System.out.println("after sleeping");
```

- Click on light bulb to show options



Unhandled exception type InterruptedException

- Add throws declaration
- Surround with try/catch

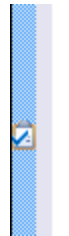
```
...  
public class Exceptions {  
    public static void main(String[] args) throws InterruptedException {  
        System.out.println("before sleeping");  
    }  
}
```

- Add throws declaration (defer error handling)
- Surround with try/catch (handle error now - **BEST**)



# Checked exceptions

- Output can be better.



```
try {  
    Thread.sleep(4321);  
} catch (InterruptedException e) {  
    // TODO Auto-generated catch block  
    e.printStackTrace();  
}
```

- Trim comment, show just message.

```
try {  
    Thread.sleep(4321);  
} catch (InterruptedException e) {  
    System.out.println(e);  
}
```



# Exercise



- Do the Thread.sleep(####) example



# Multiple catch blocks

- `try {...}`
- `catch (MinorExceptionType e) { ... }`
- `catch (AnotherMinorType e) { ... }`
- `catch (MajorExceptionType e) { ... }`
- `catch (Exception e) { ... }`
  - will catch anything else



# Defining your own error states

- Reusing Java's exception classes
  - if (you don't like something) {
    - `throw` new Exception("I didn't like it");
  - }
  - if (it's a new kind of Arithmetic problem) {
    - `throw` new ArithmeticException("My arithmetic error #3543");
  - }

# Defining your own error states



- Creating your own class of error types
  - `class MySpecialException extends Exception { }`
  - `void myLogic( ) throws MySpecialException {`
    - `if (it's a special case) {`
      - `throw new MySpecialException("I especially didn't like it");`
    - `}`



# Rethrowing

- Not common
  - try { ... }
  - catch (Exception e) {
    - System.out.println("Exception was caught.");
    - **throw** e;
  - }
- Method that calls this code **must** be in a try block.



# finally

- Code block to execute when try **OR** catch has executed.
- Must be after catch blocks
- Used for
  - cleaning up resources
    - close database connections
    - close file handles
  - finalizing processes



# Defer error handling

```
try { Aclass.doA( ); } catch (AnException a) {...}
```

```
class Aclass {  
    static void doA ( ) throws AnException {  
        BClass.doB( ); } }  
    ↗
```

```
class Bclass {  
    static void doB() throws AnException {  
        throw AnException( ); } }  
    ↗
```



# Throwable

- Exception extends Throwable
- Methods to use in **catch** block on object caught
  - `e.getMessage( );`
  - `e.toString( );`
  - `e.printStackTrace( );`





# Inheritance structure

- java.lang.Throwable
  - **Error** – serious problems, don't try to **catch**.
    - IOException, ThreadDeath, VirtualMachineError...
    - doesn't require **throws**
  - Exception
    - [AclNotFoundException](#), [ActivationException](#), [AlreadyBoundException](#), [ApplicationException](#), [AWTException](#), [BackingStoreException](#), [BadAttributeException](#), [BadBinaryOpValueExpException](#), [BadLocationException](#), [BadStringOperationException](#), [BrokenBarrierException](#), [CertificateException](#), [ClassNotFoundException](#), [CloneNotSupportedException](#), [DataFormatException](#), [DatatypeConfigurationException](#), [DestroyFailedException](#), [ExecutionException](#), [ExpandVetoException](#), [FontFormatException](#), [GeneralSecurityException](#), [GSSEException](#), [IllegalAccessException](#), [IllegalClassFormatException](#), [InstantiationException](#), [InterruptedException](#), [IntrospectionException](#), [InvalidApplicationException](#), [InvalidMidiDataException](#), [InvalidPreferencesFormatException](#), [InvalidTargetObjectTypeException](#), [InvocationTargetException](#), [IOException](#), [JAXBException](#), [JMEException](#), [KeySelectorException](#), [LastOwnerException](#), [LineUnavailableException](#), [MarshalException](#), [MidiUnavailableException](#), [MimeTypeParseException](#), [MimeTypeParseException](#), [NamingException](#), [NoninvertibleTransformException](#), [NoSuchFieldException](#), [NoSuchMethodException](#), [NotBoundException](#), [NotOwnerException](#), [ParseException](#), [ParserConfigurationException](#), [PrinterException](#), [PrintException](#), [PrivilegedActionException](#), [PropertyVetoException](#), [RefreshFailedException](#), [RemarshalException](#), [RuntimeException](#), [SAXException](#), [ScriptException](#), [ServerNotActiveException](#), [SOAPException](#), [SQLException](#), [TimeoutException](#), [TooManyListenersException](#), [TransformerException](#), [TransformException](#), [UnmodifiableClassException](#), [UnsupportedAudioFileException](#), [UnsupportedCallbackException](#), [UnsupportedFlavorException](#), [UnsupportedLookAndFeelException](#), [URISyntaxException](#), [UserException](#), [XAException](#), [XMLParseException](#), [XMLSignatureException](#), [XMLStreamException](#), [XPathException](#)



# Try-with-resources – JDK 7

- Use with resources implementing `AutoCloseable`
- Used in i/o
- Will close opened resources
- try (**managed resource**) {
  - ...
- } // managed resource is closed



# Multi-catch – JDK 7

- separate exception types with | (OR operator)
- `catch(ArithmeticException |  
ArrayIndexOutOfBoundsException) {`
  - `...`
- `}`



# Exercise – custom exceptions

- CreditCardProcessor
  - main( ) – call process
- CreditCardException
  - String creditCard
  - All constructors from superclass
- CreditCard
  - constructor
    - Calls set creditCard, throws Exception when rule fails



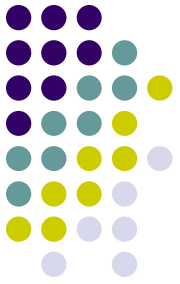
# javadoc tags

- @throws, @param, @return
- @author, @see, @since, @version
- {@code}, {@docroot}, {@inheritDoc}, {@link},  
{@linkplain}, {@literal}, {@value}
- @deprecated



# javadoc - Eclipse

- Project > Generate javadoc
  - Generate javadoc command
    - Point to a JDK bin directory location of the javadoc utility program
  - Select packages
  - Select visibility
  - Select doclet (formatter: standard = html)



# END TASKS



# .jar files

- File > Export...
- Java > JAR file > Next
  - JAR file is self-executing if you choose
  - Runnable JAR file - for self-executing jars with GUIs using a launch configuration
- Select the project (src only)
- Check: Export Java source files and resources
- Select export destination: name of .jar file
- Options: compress





# 105 End tasks

- Zip files and save/mail
- Certificate
- Evaluation