Latest Syntax Enhancements

Topic Covered

Java 5

- Generics
- For-each Loops
- Varargs
- Autoboxing
- Enums
- Static Import
- Annotations

Java 7

- Strings in switch
- ARM
- Diamond for generics
- Underscore for numeric literals
- Binary literal
- Multi-catch

Java 5

Syntax Enhancements

Generics

- Allows type safe runtime environment

- The goal:
  - If ***javac -source 1.5*** raises no unchecked warnings – then the application is type-safe

Generics

- Taking out an element from a collection might be
  - Unsafe – compile-time can’t predict wrong Object casting
  - Unnecessary – when the contained type is known
- Specifying the collection type helps in
  - Checking on compile-time
  - Perform automatic casting on run-time

Generics

```
public void clearLongStrings(Collection c) {
    for (Iterator i = c.iterator(); i.hasNext(); )
      if (((String) i.next()).length() == 4)
        i.remove();
}
```

```
public void clearLongStrings(Collection<String> c) {
    for (Iterator<String> i = c.iterator(); i.hasNext(); )
      if (i.next().length() == 4)
        i.remove();
}
```

Generics

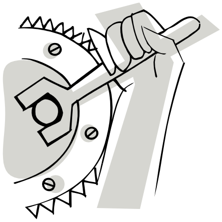
- Creating type specific collections

```
Set<String> s = new HashSet<String>()

or

Set<String> s = Collections.checkedSet(mySet, String.class);
```

- Subclasses of the specified type may be assigned
- Any illegal assignment will throw *ClassCastException*

Exercise

Lab 1



- *Employee* class describes an employee and has the following data members: first, last, salary and department.
- *EmployeeStatistics* gives statistic information for any *Employee* collections provided. Since it is an utility class, all its methods are static.
- Test class initiates an *Employee* collection (*ArrayList*) and calls *EmployeeStatistics* to gather some statistic information.

In this lab you are required to do the following:

- Code 3 type-safe methods in *EmployeeStatistics*:
  - define a type-safe static method named **averageSalary** that takes an *ArrayList* of Employees and calculates the average salary and returns it
  - define a type-safe static method named **numOfEmployees** that takes an *ArrayList* of Employees and returns the number of employees in the list
  - define a type-safe static method named **numOfEmployees** that takes an *ArrayList* of Employees and department name and returns the number of employees in the specified department
- In *Test* class – call the new methods in order to print statistic data

Note: the collection used in the labs is ***ArrayList***

Generics

- Generics & sub-typing

```
List<String> ls = new ArrayList<String>();
List<Object> lo = ls;       // fails to compile

lo.add(new Object());
String s = ls.get(0);       attempts to assign an Object to a String!
```

- Even if *String* extends *Object*

  Still, *E<String>* is not a subtype of *E<Object>*

- E<Object> means that only Object collections may be assigned

Generics

- Wildcards

```
public void printCollection (Collection c) {
   //prints a heterogenic collection
}
```

```
public void printCollection (Collection<Object> c) {
   //efficient print of an Object type-safe collection
}
```

```
public void printCollection (Collection<?> c) {
   //efficient print of a heterogenic type-safe collection
}
```

- All methods in this case has the same signature
  - All takes a Collection
  - All named ‘printCollection’
  - Will cause method collision if written in the same class

Generics

Wildcards

- <?> stands for unknown type

```
public void printCollection (Collection<?> c) {
   //efficient print of a heterogenic type-safe collection
}



//Calling this method can be done like that:

Collection<Object> col1=new Vector<Object>();
printCollection(col1);

Collection<String> col2=new Vector<String>();
printCollection(col2);
```

Generics

- Another look of wildcard type:

```
//any type-safe collection may be assigned:
Collection<?> c = new ArrayList<String>();

//but, once assigned – it becomes type-safe specific:
c.add(new Object()); // compile time error
```

Generics

Wildcards

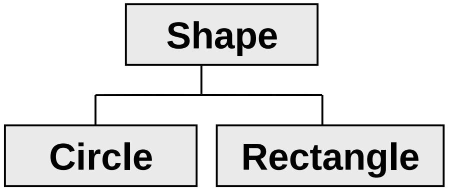
- Nothing but null can be assigned to a Collection<?>

```
Collection<?> c = new ArrayList<?>();
c.add(new Object()); // compile time error
c.add(new String(“hello”)); // compile time error
c.add(null); // ok
```

Generics

- Bounded Wildcards

```
public void drawAll(List<Shape> shapes) {
      for (…..) {
            s.draw(this);
      }
}
// but, Shape is an abstract class
// Circle or Rectangle type-safe collections will cause exception
```



Shape

Circle

Rectangle

```
public void drawAll(List<? extends Shape> shapes) {
      for (…..) {
            s.draw(this);
      }
}
// this is the right way of assigning Shape and its subclasses
```

Generics

- Bounded Wildcards

```
Collection<? extends Shape> c = new Collection<? extends Shape>();
// compile time error
```

- <?> stands for unknown type

- Should be done like this:

```
Collection<Shape> c = new Collection<Shape>();
c.add(new Rectangle());
```

Generics

- The compiler doesn’t know the relationship between <?> and Shape.

```
Collection<? extends Shape> c = new Collection<Shape>();
// only null values can be assigned – not so useful…
```

- Using <? extends …> is good for ‘read only’

- <? extends Interface> is also supported

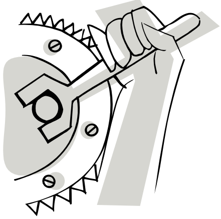
```
public void saveIt (Collection<? extends Serializable> col){
            …..
}
```

Generics

- <? super Class> - for specifying any super-class type-safe entity:

```
public void check (Collection <? super Manager> col){
          …
}
```

Exercise



Lab 2

- *Manager* class was added. *Manager* extends *Employee*.
- In Test class the *Employee* collection is now heterogenic and contains both *Employees* and Managers.
- When trying to assign the new heterogenic collection to *EmployeeStatistics* a compilation errors are raised.

In this lab you are required to do the following:

- Fix all compilation errors by changing code so statistics made by the utility methods will support heterogenic *Employee* collections.

Generics

- Generic methods
  - Automatic result casting
  - Compile-time arguments check
- When to use ?
  - When there is a linkage or dependency between method parameters and return types
  - otherwise – use wildcards

Generics

- Generic methods - example

```
public <T> void copy(List<? extends T> src, List <T> dest) {
    …
}
```

or

```
public <T, S extends T > void copy (List<S> src, List <T> dest) {
    …
}
```

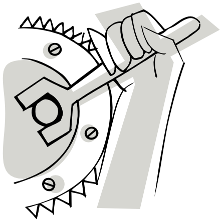
Generics

- Another example emphasizes the differences between extends and super:

```
public <T> void copy(List<? extends T> src, List <T> dest) {
    for (int i=0;i<src.size();i++)
            dest.add(src.get(i));
}
```

```
public <T> void copy (List<T> src, List <? super T> dest) {
    for (int i=0;i<src.size();i++)
              dest.add(src.get(i));
}
```

Exercise



Lab 3

- More support is required when dealing with Managers and heterogenic collections.

In this lab you are required to do the following:

- Code 2 more type-safe methods in *EmployeeStatistics*:
  - write a static method named **getManagers** that will take a type-safe *Employee ArrayList*

    and returns a type-safe Manager *ArrayList* with all the managers
  - write a static generic method named **insertEmployees** that takes :

    source - which is any type-safe *ArrayList* contains objects that extends *Employee* (like <*Manager*>)

    destination - which is a type-safe *Employee ArrayList*

    the method inserts the source into the destination and returns void
- Test class is already fully coded to use and call the new methods

Generics

- Compile–Time errors
  - Occurs when the translated code uses wrong casting
  - Occurs when assigning objects to <?> base type

- Compile-Time unchecked warnings
  - Occurs when the compiler has no way of insuring types
  - Means that the code has potential run-time errors

```
public String insert(Integer x) {
    List<String> ys = new LinkedList<String>();
    List xs = ys;
    xs.add(x); // compile-time unchecked warning
    return ys.iterator().next();
}
```

Generics

- *instanceof* operator
  - Generics check are not supported

```
Collection cs = new ArrayList<String>();
if (cs instanceof Collection<String>) { ...} // illegal
```

- Casting
  - When can’t be checked – will result in warning

```
Collection<String> cstr = (Collection<String>) cs; // unchecked warning

public <T> T badCast(T t, Object o) {
		return (T) o; // unchecked warning
}
```

Generics

- java.lang.Class
  - Is a generics supported class
  - Class<T> - <T> stands for the represented class
  - For example:
    - the type of *String.class* is Class<String>
    - the type of *Serializable.class* is Class<Serializable>
  - The *newInstance()* method returns T

```
String s=“Hello";
Class<? extends String> c=s.getClass();
String st=c.newInstance();  //no casting is needed
```

For-Each Loop

- Iterating over a collection is ugly

```
public void paySalary (Collection<Employee> emp){
        for (Iterator<Employee> iterator=emp.iterator();iterator.hasNext();)
                  iterator.next().pay();
}
```

- For-Each loop makes it look much better:

```
public void paySalary (Collection<Employee> emp){
        for (Employee curr : emp)
                  curr.pay();
}
```

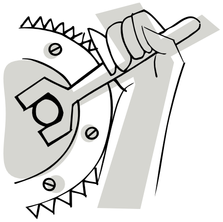
For-Each Loop

- Array of objects or primitives

```
public int sumArray (int[] nums){
        int sum=0;
        for (int i : nums)
                  sum+=i;
        return sum;
}
```

```
public double concat (String[] words){
        String sentence=“”;
        for (String curr : words)
                  sentence+=curr+” “;
        return sentence;
}
```

Exercise



Lab 4

In this lab you are required to do the following:

- In *EmployeeStatistics* class, change all index loops into for- each loops.

Varargs

- Allows multiple type-safe parameters assignment to a method as units

```
public int sum (int… numbers){
        int sum=0;
        for (int x : numbers)
                  sum+=x;
                  retrun sum;
}
```

```
Varargs usage :

int total = sum(10, 45, 88, 90);
```

Varargs

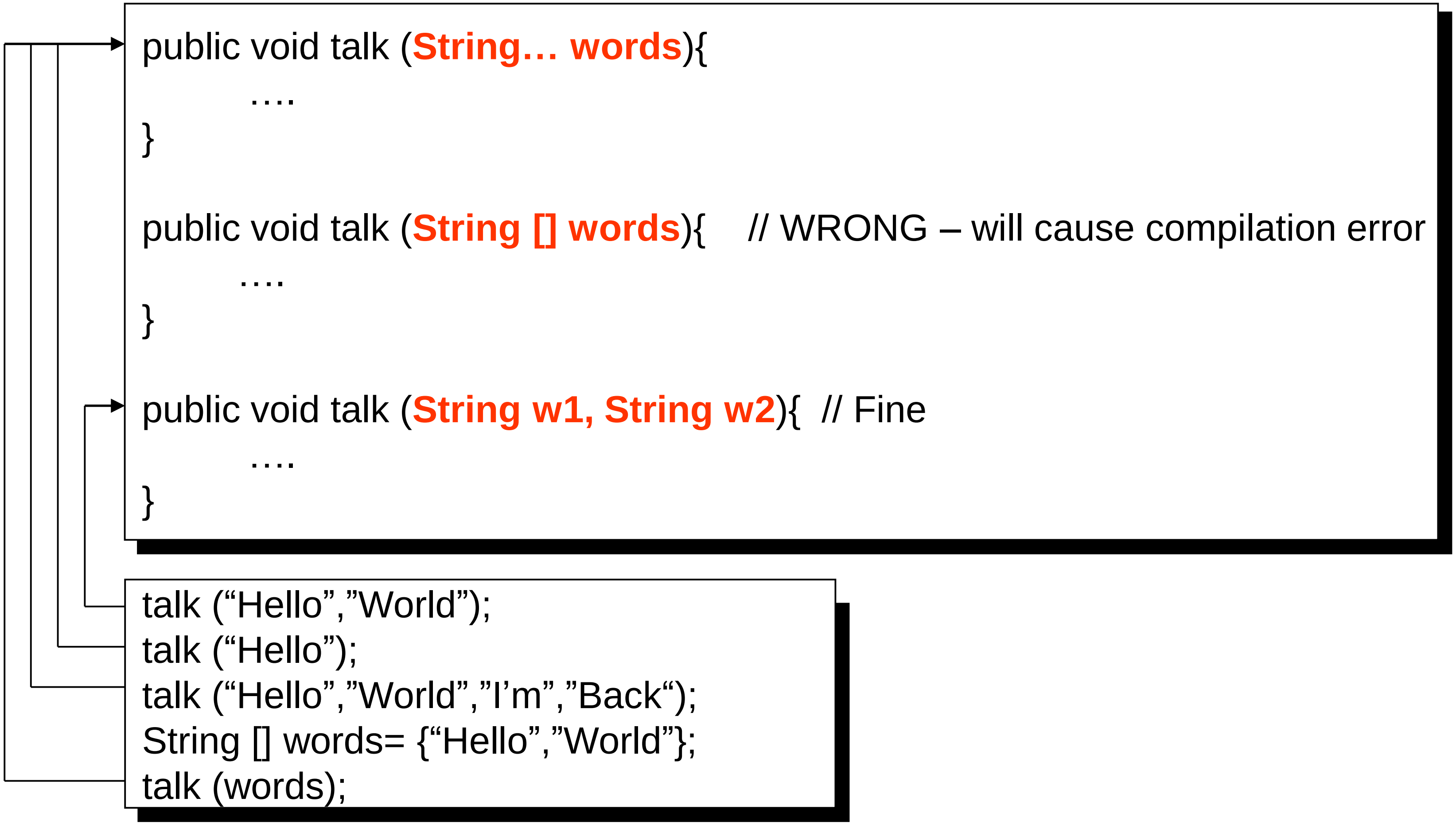
- Method overloading issue

  - Varargs equals to an array
  - Therefore:
    - Cannot be overloaded with a method that takes an array
    - If it is not the only parameter – varargs must be the last one
  - Arrays can be also assigned as a varargs
  - main method – new look:

```
public static void main (String… args){
        …..
}
```

Varargs

- Examples:



```
public void talk (String… words){
          ….
}

public void talk (String [] words){    // WRONG – will cause compilation error
         ….
}

public void talk (String w1, String w2){  // Fine
          ….
}
```

```
talk (“Hello”,”World”);
talk (“Hello”);
talk (“Hello”,”World”,”I’m”,”Back“);
String [] words= {“Hello”,”World”};
talk (words);
```

Varargs

- Examples:

```
public void talk (String… words){
         ….
}

public void talk (int x, String word, String… words){  // Fine
        ….
}

public void talk (String… words, String word){  // WRONG  - will cause compilation error
         ….
}
```

Autoboxing

- Inboxing - taking a primitive and wrap it in an object
- Outboxing - getting a wrapped primitive value out of an object
- Done a lot in Java wrapper classes (like Integer)

```
int num = 100;
Integer i = new Integer(num);

int other = i.intValue();
```

- Autoboxing – means you don’t need to do it anymore!

Autoboxing

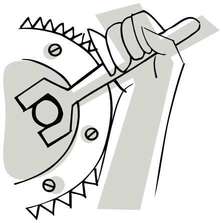
- Example :

```
public class IntMaster {

        private int[] nums = {1,2,3,4,5,6,7,8,9,10};

        public Integer getInt(int index){
                return nums[index];
        }

        public void setInteger (Integer toReplace, int index){
                nums[index] = toReplace;
        }
}
```

Autoboxing

- Remember :
  - Boxing is far from being efficient
  - Use it only to contain primitives in an object collection
  - Never use it for scientific calculations

Exercise



Lab 5

- More reports are needed

In this lab you are required to do the following:

- Code 1 more type-safe method in *EmployeeStatistics*:
  - write a type-safe method named **getManPowerReport** that takes an *ArrayList* of *Employees* and returns a type-safe

    HashMap that contains:

    "managers" as key, and the actual number of managers in the collection as value

    "employees" as key, and the actual number of employees in the collection as value.

- Test class is already fully coded to use and call the new methods

Enums

- Understanding enumeration types:
    - Specify customized types
    - Define optional values

    - Currently done like that:

```
public static final int STATE_AVAILABLE=0;
public static final int STATE_AWAY=1;
public static final int STATE_OFFLINE=2;
```

Enums

- So, what’s wrong with current implementations ?
  - Not type-safe

```
int currentState = 25;
currentState = STATE_AWAY + STATE_OFFLINE;
```

  - No namespace – all state options should have the State prefix

```
public static final int STATE_AVAILABLE=0;
public static final int STATE_AWAY=1;
public static final int OFFLINE=2;
```

  - Brittleness – changing values will require client compilation

```
STATE_AVAILABLE=0;
STATE_AWAY=1;
STATE_OFFLINE=2;
```

```
STATE_AVAILABLE=0;
STATE_AWAY=1;
STATE_BLOCKED=2;
STATE_OFFLINE=3
```

Enums

- J2SE 1.5 has a built in enum types support

```
public class Client {

        public enum State {AVAILABLE, AWAY, OFFLINE}

        private State currState = null;  //null assignment is allowed

        public Client () {
              currState=State.OFFLINE;
        }
        ….
}
```

Enums

- Some features of enums
  - *toString()* of enums returns it represented value

```
public class Client {

        public enum State {AVAILABLE, AWAY, OFFLINE}

        private State currState;

        public Client () {
                currState=State.OFFLINE;
        }
        public void printState(){
                System.out.println(currState);    // 'OFFLINE' is printed
        }
}
```

Enums

- Some features of enums
  - printing

```
public class Client {

        public enum State {AVAILABLE, AWAY, OFFLINE}

        private State currState;

        public Client () {
              currState=State.OFFLINE;
        }
        public void printOfflineState(){
              System.out.println(State.OFFLINE);    // ‘OFFLINE’ is printed
        }
}
```

Enums

- Some features of enums
  - *ordinal()* prints the index of the current enum value

```
public class Client {
    //ordinal              0       1       2
    public enum State {AVAILABLE, AWAY, OFFLINE}

    private State currState;

    public Client () {
        currState=State.OFFLINE;
    }
    public void printOrdinal(){
        System.out.println(currState.ordinal());    // ‘2’ is printed (0 is AVAILABLE, 1 is AWAY)
    }
}
```

Enums

- Some features of enums
  - *equals()* checks enums according to its constants

```
public class Client {

        public enum State {AVAILABLE, AWAY, OFFLINE}

        private State currState;

        public Client () {
                currState=State.OFFLINE;
        }
        public boolean isOffline(){
                if(currState.equals(State.OFFLINE)) //or: (currState.compareTo(State.OFFLINE)==2)
                        return true;
                return false;
        }
}
```

Enums

- Some features of enums
  - *values()* method returns the list of enum values

```
public class Client {

        public enum State {AVAILABLE, AWAY, OFFLINE}

        private State currState;

        public Client () {
              currState=State.OFFLINE;
        }
        public void printStateList(){
              for(State state : State.values())
                            System.out.println(state);
        }
}
```

Enums

- Calling inner Enums from outside the class

```
public class Client {

        public enum State {AVAILABLE, AWAY, OFFLINE)}

        …
}
```

Using enum from outside Client class:

```
    enums.Client.State  s = enums.Client.State.AWAY;
```

Enums

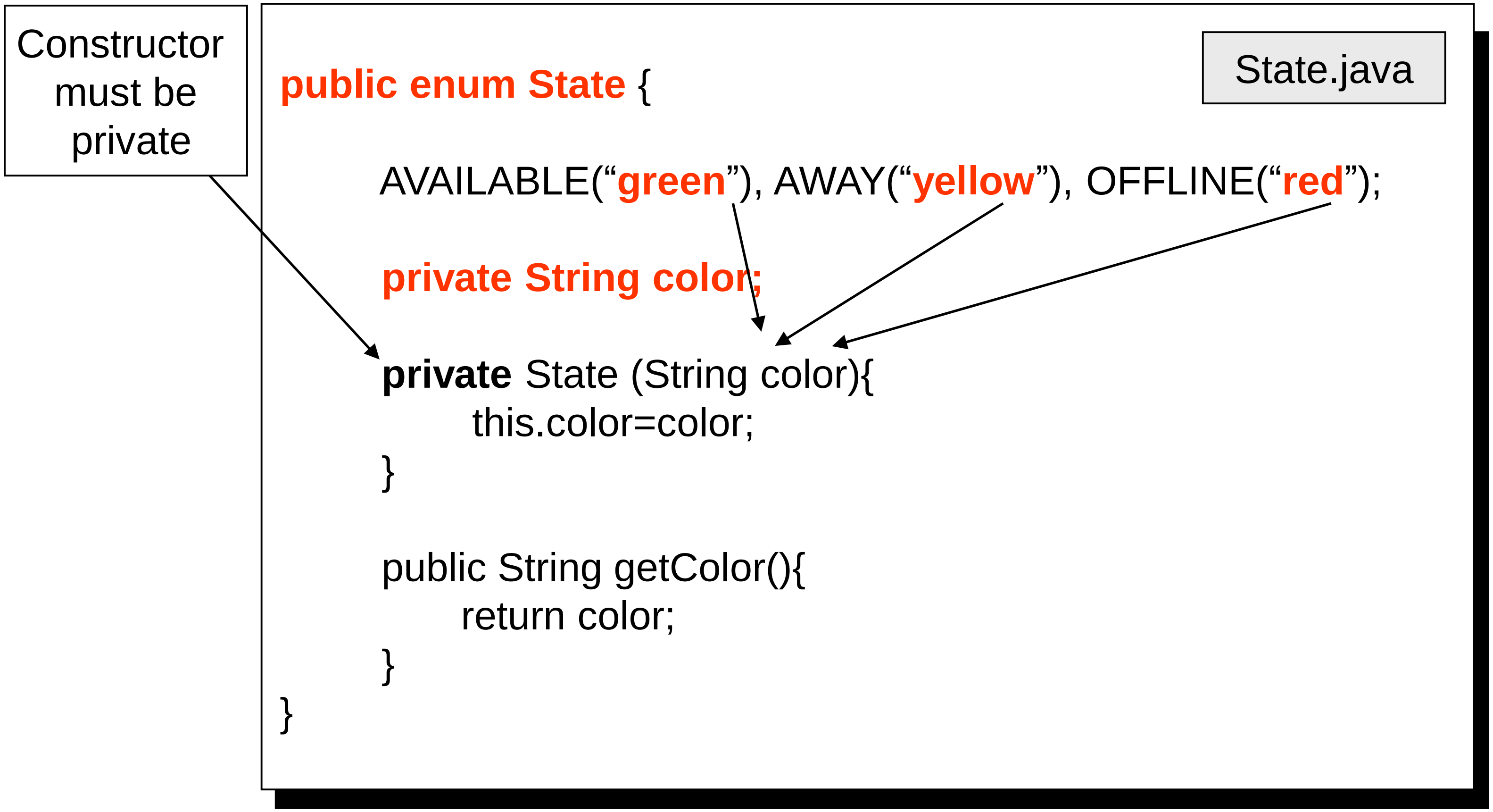
- Some features of enums
  - Using enums in switch block

```
public class Client {

        public enum State {AVAILABLE, AWAY, OFFLINE)}

        private State currState;
        …
        public void setClientState(){
              switch (currState){
                        case AVAILABLE: //set client to available state
                                          break;
                        case AWAY: //set client to away state
                                          break;
                        case OFFLINE: //set client to offline state
                                          break;

              }
        } …
```

Note that Java knows the enum type of the switch cases since *currState* is a State enum type

Enums

- Enums may hold additional data & methods



Constructor must be private

State.java

```
public enum State {

        AVAILABLE("green"), AWAY("yellow"), OFFLINE("red");

        private String color;

        private State (String color){
                this.color=color;
        }

        public String getColor(){
                return color;
        }
}
```

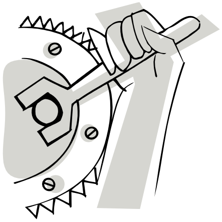
Enums

- Some points to remember:
  - Enums cannot be inherited
  - Enums constructor cannot be invoked programmatically (Done only by the compiler)
  - All Enums are of type java.lang.Enum

```
….
Enum e = State.AWAY;
…
String name=e.name();
int index=e.ordinal();
Class<State> class=e.getDeclaredClass();
…
```

  - clone() isn’t supported – throws *CloneNotSupportedException*

Exercise



Lab 6

- The use of standard and consistent manager ranks is to be added to the application environment

In this lab you are required to do the following:

- Create new enum called ***Rank*** (in a file *Rank.java*) and define the next values:
  - MANAGER
  - DIRECTOR
  - VICE_PRESIDENT
  - PRESIDENT
- Update *Manager* class to use the *Rank* enum instead of *Strings*
- Code 1 more type-safe method in *EmployeeStatistics*:

  write a type-safe method named **getManagerRanks** that takes an *ArrayList* of *Employees* and returns a type-safe *HashMap* that contains:
  - *Manager* instance as key
  - *Rank* enum as value
- In *Test* class update the *Manager* initiation to use *Rank* enum instead of *String* values
- The line that calls the *getManagerRanks (..)* is already called

Static Import

- Instead of doing that:

```
public double calculate(double startValue){
        return startValue*Math.PI+100/Math.E;
}
```

- Programmers prefer Constant Interfaces:

```
public interface MyConstants{
      public double PI = 3.141592653589793;
      public double E= 2.718281828459045;
}
```

```
public class MyClass implements MyConstants{
…
public double calculate(double startValue){
        return startValue*PI+100/E;
}
```

Static Import

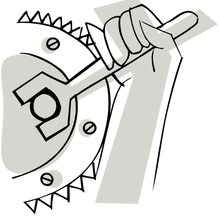
- Constant Interface Anti-pattern
  - Ease of use shouldn’t have structural influence
  - Class that implements an interface must take it all
  - The polymorphic ability that gained is irrelevant
- In other words, this is not a good solution

Static Import

- The solution – Static Imports
  - Import static members and static methods only
  - Allows unqualified access to static member of other class/interface
  - Done without inheriting the content of the other class/interface

```
import static java.lang.Math.*;
or
import static java.lang.Math.PI;
import static java.lang.Math.E;

public class MyClass{
…
public double calculate(double startValue){
        return startValue*PI+100/E;
}
```

Exercise



Lab 7

A static member is required in order to check the number of *Employee* instances in memory.

In this lab you are required to do the following:

- In *Employee* class:
  - add a static member named EMPLOYEE_COUNT
  - Update the constructor to increase EMPLOYEE_COUNT on creation
- In *Test* class:
  - Perform a static import to *Employee* class
  - Print EMPLOYEE_COUNT after collection initiation

Note: a package is required for the compiler to map the *Employee* class location.

Therefore, all classes are members of ‘application‘ package.

In order to compile use the –d parameter (javac –d . *.java)

In order to run Test use this command: java application.Test

Annotations / Metadata

- Currently, many API's requires extra code and files:
  - JAX-RPC – requires interface & implementation
  - Java Beans – requires *BeanInfo* class
  - EJB – requires DD (*ejb-jar.xml*)
  - Web Applications – requires DD (*web.xml*)
  - *transient* modifier – required to specify un-saved values
  - *@deprecated* modifier – required to specify un-used methods
- Annotations are to hold that extra data as classes.
- Annotations doesn't effect program logic.
- But they do effect the way program treated by tools & libraries.

Annotations

- First, annotation structure must be defined

- There are several types of annotations:
  - Empty annotations – are used to sign classes
  - Single value annotations – for example – copyright annotation
  - Multi values annotations – usually holds configuration info

- Annotation’s elements might be:
  - *java.lang.String*
  - Primitives
  - *java.lang.Class*
  - Enums
  - Annotations
  - Arrays of all the above

Annotations

- Empty annotations

- Defining empty annotation

```
public @interface ThisClassIsMine { }
```

- Attaching empty annotation to a specific class

```
@ThisClassIsMine public class Employee{
    ….
```

Annotations

- Single value annotations

- Defining single value annotation

```
public @interface Copyright {
        String value();
 }
```

The single element usually named ‘*value*’*, with type of String*

- Attaching single value annotation to a specific class

```
@Copyright (“2005 John Bryce Training Center”)
public class Employee{
   ….
```

Annotations

- Multi values annotations
- Defining multi values annotation

```
public @interface ClientConfiguration {
     int id();
     String ip();
     String port();
     State state(); //enum
}
```

- Attaching multi values annotation to a specific class

```
@ ClientConfiguration(
     id=12345,
     ip=“127.0.0.1”,
     port=5555,
     state=State.OFFLINE)
public class Client {
    ….
```

Annotations

java.lang.annotation package

- Specifies the super interface of all annotations
- Provides some pre-defined helper annotations:

  (are used to define other annotations)

  - *Documented* – the annotation should appear in javadoc
  - *Inherited* – the annotation is inherited to subclasses
  - *Retention* – specifies the scope of the annotation
  - *Target* – specified the types that annotation can be used in

Annotations

Some more regarding pre-defined annotations:

- *Target* – values might be:

Possible values specified as static contstants of *ElementType* class:

TYPE – for classes, interfaces & enums
ANNOTATION_TYPE – for other annotations
CONSTRUCTOR
FIELD
METHOD
LOCAL_VARIABLE
PACKAGE
PARAMETER – method parameters

- Is a single value annotation
- Default – all

```
@Target(value={TYPE,FIELD,METHOD,PARAMETER,CONSTRUCTOR,LOCAL_VARIABLE})
```

Annotations

Some more regarding pre-defined annotations:

- *Retention* – values might be:

Possible values specified as static contstants of *RetentionPolicy* class:

CLASS – means that the compiler will store the annotation in the generated class – but are not used by the VM at runtime

RUNTIME – means that the compiler will store the annotation in the generated class and that it will be used by the VM at runtime, usually via reflection

SOURCE – means that the compiler will discard the annotation

- Is a single value annotation
- Default - CLASS

Annotations

Some more regarding pre-defined annotations:

- *Documented* – annotation will appear in the API docs
  - Target: ANNOTATION_TYPE
  - Retention: RUNTIME
  - Is an empty annotation
- *Inherited* – annotation will be passed to subclasses
  - Annotations are automatically inherited – but their values are not
  - Inherited annotation will cause the values to be loaded from super-classes when not available in the current class
  - Target: ANNOTATION_TYPE
  - Retention: RUNTIME
  - Is an empty annotation

Annotations

- Using pre-defined annotations to declare annotations

```
import java.lang.annotation.*;

@Target (ElementType.TYPE)
@Retention (RetentionPolicy.RUNTIME)
public @interface ClientConfiguration {
        int id();
        String ip();
        String port();
        State state(); //enum
 }
```

Annotations

There are some annotations in java.lang package:

- *Deprecated* – indicates deprecated entities
  - Target: all
  - Retention: RUNTIME
  - Is an empty annotation
- *Override* – indicates method override
  - Target: METHOD
  - Retention: SOURCE
  - Is an empty annotation
- *SupressWarnings* – generate compile-time warnings
  - Target: TYPE, FIELD, CONSTRUCTOR, METHOD, LOCAL_VARIABLE
  - Retention: SOURCE
  - Is a single value annotation, the value holds the warning

Annotations

Working with annotations in runtime

- *java.lang.reflect* has annotation support
- Every reflected entity has these 3 methods:

```
public <T extends Annotation> T getAnnotation (Class< T extends Annotation > annotationClass)

public Annotation [ ] getAnnotations()

public Annotation [ ] getDeclaredAnnotations()
```

Annotations

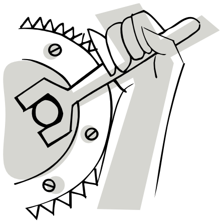
- An example:

```
public @interface Copyright {
          String value();
 }
```

```
@Copyright (“2005 John Bryce Training Center”)
Or
@Copyright (value=“2005 John Bryce Training Center”)
public class Employee{
    ….
```

```
..
public static void main(String args) {
    try{
         Copyright copyright = Class.forName(“Employee”).getAnnotation(Copyright.class);
         System.out.println(copyright);
    }catch (ClassNotFoundException e){…}
}..
```

Exercise



Lab 8

Copyrights are needed in business classes *Employee* & *Manager*

In this lab you are required to do the following:

- Create annotation type named *WritersRights* (*WritersRights.java*) with the following:
  - Single value
  - Retention - runtime
- Update both *Employee* and *Manager* classes to use *WritersRights* annotation
- Run Test class which enquires *Employee* & *Manager* and displays the *WritersRights* annotation values

Java 7

Syntax Enhancements

Syntax Enhancements

- Strings in switch

```
String value=“one”;
….
switch(value){
      case “one”: ………
      case “two”: ………
      default: …….
}
```

- It’s about time…

Syntax Enhancements

- ARM – Automatic Resource Management

  - Opening / closing resource connection is not part of the try-catch block
  - Instead of:

```
public void doIO() throws IOException{
    FileInputStream in=null;
    try{
        in=new FileInputStream (“file”);
        int data = in.read();
    }catch(FileNotFoundException e){
        in.close();
    }
}
```

  - We use:

```
public void doIO() throws IOException{
    try(FileInputStream in= new FileInputStream (“file”)){
        int data = in.read();
    }
}
```

  - Forces the resource to be “Auto Closable”

Syntax Enhancements

- ARM – Automatic Resource Management
  - Closable.close() method throws IO exception
  - In order to use ARM for other APIs as well – an AutoClosable super interface was created
    - AutoCloseable close() method throws a generat Exception
    - Closeable now extends it
    - JDBC API is now AutoClosable just like IO

```
public  interface AutoClosable {
                public void close () throws Exception;
}
```

```
public  interface Closable extends AutoClosable {
                public void close () throws IOException;
}
```

Syntax Enhancements

- More on ARM
  - Manages “AutoCloseable” implementations only(!)
  - Whether try block pass or fails – close() will be invoked
  - Can declare and use more than one resource:

```
try(FileInputStream in= new FileInputStream (“file1”);
     FileOutputStream out= new FileOutputStream(“file2”) ){
            int data = in.read();
            out.write(data);
}
```

  - Close() method is called according to resource declaration order in the try clause

Syntax Enhancements

- Improved generic type creation
  - Instead of:

```
Map<String,List<Integer>> map=new Map<String,List<Integer>>();
```

  - We use:

```
Map<String, List<Integer>> map=new Map<>();
```

- Binary literals
  - We already have 0 (octal) & 0x (hexadecimal)
  - Now we have 0b (binary) as well:

```
int binary = 0b11011101;
```

- Underscores for numeric literals

```
int million =1_000_000;
```

Syntax Enhancements

- Multi-catch
  - Relating to different exceptions in a single catch block
  - Instead of:

```
try{
        FileInputStream in=new FileInputStream (“file”);
        Connection con = DriverManager.getConnection(….);
        ….in.read();
        ….con.createStatement();
}catch(IOException e){
        ….
}catch(SQLException e){
        ….
}
```

  - We use:

```
try{
        …..
}catch(IOException | SQLException e){ …. }
```