Zaawansowane programowanie obiektowe

Lecture 2

Szymon Grabowski
sgrabow@kis.p.lodz.pl
http://szgrabowski.kis.p.lodz.pl/zpo18/

Łódź, 2018

clone() – you shouldn’t (mostly) use it

[based on J. Bloch, *Effective Java*, 3rd ed., Item 13]

Many problems with the clone method from Object (e.g., the clone architecture is incompatible with normal use of final fields referring to mutable objects)!

Alternatives:

provide a copy constructor or copy factory.

A copy constructor is a constructor that takes a single argument whose type is the class containing the constructor, for example:

```
public X(X x);
```

A copy factory is the static factory analog of a copy constructor:

```
public static X newInstance(X x);
```

Interfaces

An interface is an abstract class which
has only abstract methods.
Using an interface: a class promises to implement
all the methods declared in a given interface.

Bloch: *an interface is generally the best way to define a type that permits multiple implementations*.

```
interface Barkable {
    void bark();
    void barkLoud();
    void howl();
    void whine();
}
```

```
class Dog implements Barkable {
    @Override
    public void bark() { ... }
    .....
}
```

Interfaces, cont'd

Interfaces declare public abstract methods
(hence the words public, abstract are redundant).
But *public* in front of the method from an interface is required in the implementing class.

Interfaces may (but rarely should!) also store
static constant fields.
(Again, the words *final static* are not necessary.)

The public interface CharSequence is implemented by
CharBuffer, String, StringBuffer, StringBuilder.

A CharSequence is a readable sequence of characters.
This interface provides uniform, read-only access to
many different kinds of character sequences. Its methods:

```
char charAt(int index)
int length()
CharSequence subSequence(int start, int end)
String toString()
```

Adapters

An adapter is a class that implements an interface, with empty definitions for all of the functions.

The idea is that if we subclass the adapter, we will only have to override the functions that we are interested in.

```
interface R2D2 {
  String display();
  String show();
  int calculate(int a, int b);
  float generate();
  double mean();
}
```

```
abstract class R2D2Adapter implements R2D2 {
  public String display() { return ""; } // say, we'll override only this
  public String show() { return ""; }
  public int calculate(int a, int b) { return 0; }
  public float generate() { return 0.0f; }
  public double mean() { return 0.0d; }
}
```

General design criteria

Protected variations

You need to isolate volatility in your application.
If you feel an application component could change,
then take steps to segregate that component using interfaces.
Interfaces will allow you to change the implementing class
without affecting existing application dependencies.

Low coupling

Ensure that changes made in one section of code
don’t adversely affect another unrelated section.
Eg. when a user interface change requires a change to the DB,
then the application is brittle (a small change
propagates throughout the software system).

High cohesion

Closely related things should be tied together tightly.

[W. Clay Richardson et al., *Professional Java, JDK 5 Edition*, Wrox, 2005]

Class design hints

[*CoreJava*, vol. 1, ch. 4]

Always keep data private.

Doing anything else violates encapsulation. Bitter experience has shown that how the data are represented may change, but how they are used will change much less frequently.
When data are kept private, changes in their representation do not affect the user of the class, and bugs are easier to detect.

Always initialize data.

Java won’t initialize local variables for you, but it will initialize instance fields of objects.
Don’t rely on the defaults, but initialize the variables explicitly, either by supplying a default or by setting defaults in all constructors.

http://e-university.wisdomjobs.com/core-java/chapter-1216-231/objects-and-classes_class-design-hints_7549.html

Class design hints, cont’d
[*CoreJava*, vol. 1, ch. 4]

Don’t use too many basic types in a class.

I.e., replace multiple related uses of basic types with other classes.
E.g., replace:

```
        private String street;
        private String city;
        private String state;
        private int zip;
```

with a new class Address.

Not all fields need individual field accessors and mutators.

You may need to get and set an employee’s salary,
but you won’t need to change the hiring date
once the object is constructed.

http://e-university.wisdomjobs.com/core-java/
chapter-1216-231/objects-and-classes_class-design-hints_7549.html

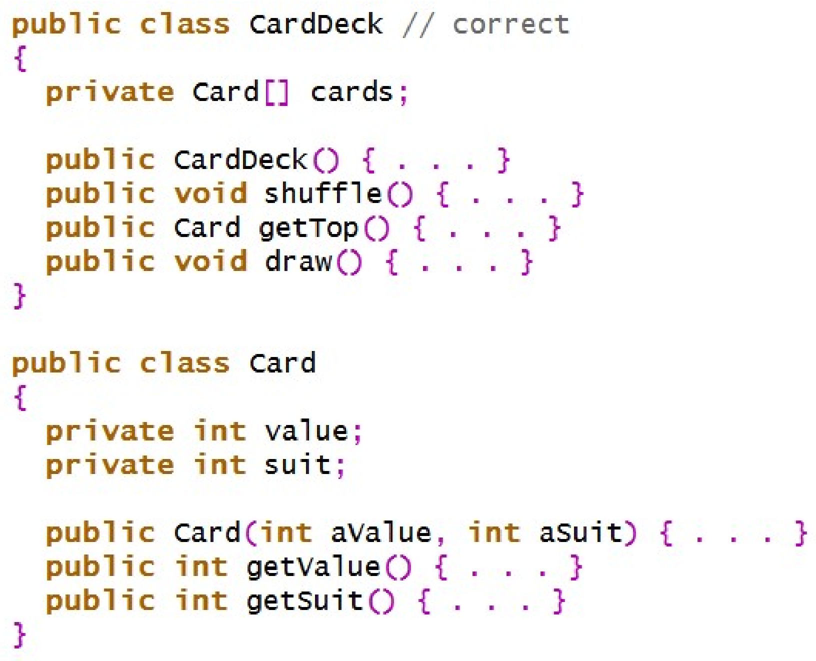
Class design hints, cont'd [*CoreJava*, vol. 1, ch. 4]

Break up classes that have too many responsibilities.

If there is an obvious way to make one complicated class
into two classes that are conceptually simpler, seize the opportunity.

```
public class CardDeck // bad design
{
        private int[] value;
        private int[] suit;

        public CardDeck() { . . . }
        public void shuffle() { . . . }
        public int getTopValue() { . . . }
        public int getTopSuit() { . . . }
        public void draw() { . . . }
}
```

Better: introduce a Card class
that represents an individual card.
Now you have two classes,
each with its own responsibilities:



```
public class CardDeck // correct
{
  private Card[] cards;

  public CardDeck() { . . . }
  public void shuffle() { . . . }
  public Card getTop() { . . . }
  public void draw() { . . . }
}

public class Card
{
  private int value;
  private int suit;

  public Card(int aValue, int aSuit) { . . . }
  public int getValue() { . . . }
  public int getSuit() { . . . }
}
```

http://e-university.wisdomjobs.com/core-java/chapter-1216-231/objects-and-classes_class-design-hints_7549.html

Is it safe?

```java
import java.util.Date;

/**
* Class invariant: start() <= end()
*/
interface TimeInterval
{
  Date getStart();
  Date getEnd();
}
```

```java
import java.util.Date;

public class TimeIntervalImpl1 implements TimeInterval
{
  private final Date start;
  private final Date end;
  public TimeIntervalImpl1( Date s, Date e ) {
    start = s;
    end = e;
    assert invariant() : "start>end";
  }

  // public void setStart(Date d) { start = d; }
  // line above doesn't compile of course
  // "cannot assign a value to final variable start" message

  public Date getStart() { return start; }
  public Date getEnd() { return end; }
  public boolean invariant()
  { return start.compareTo(end) <= 0; }
}
```

It isn’t safe

```
import java.util.*;

public class MaliciousDemo {
  public static void main(String[] args) {
    Calendar cal = new GregorianCalendar(2007, Calendar.MARCH, 8);
    Date d1 =  cal.getTime();
    cal = new GregorianCalendar(2007, Calendar.APRIL, 6);
    Date d2 =  cal.getTime();

    TimeIntervalImpl1 tii1 = new TimeIntervalImpl1(d1, d2);

    Date temp = tii1.getStart();    // !!!
    temp.setMonth(Calendar.JUNE);   // == 5 (JANUARY is 0)
    System.out.println(tii1.getStart());  // !!! MODIFIED :-(
  }
}
```

```
  // Output: Fri Jun 08 00:00:00 CEST 2007
```

11

Now it’s safe

```java
import java.util.Date;

public class TimeIntervalImpl2 implements TimeInterval
{
  private final Date start;
  private final Date end;

  public TimeIntervalImpl2( Date s, Date e ) {
    start = new Date(s.getTime());
    end = new Date(e.getTime());
    assert invariant() : "start>end";
  }

  public Date getStart() { return (Date)start.clone(); }
  public Date getEnd() { return (Date)end.clone(); }
  public boolean invariant() { return start.compareTo(end) <= 0; }
}
```

TimeIntervalImpl2 makes defensive copies in both the constructor and the accessors, which means that no outside object holds a reference to the parts of the time interval.

12

Static factory methods [J. Bloch, *Effective Java*, Item 1]

The standard way for a class to allow a client to obtain an instance is to provide a public constructor.

There is an interesting alternative though:
a public static factory method:
a static method that returns an instance of the class.

```
public static Boolean valueOf(boolean b)  {
    return (b ? Boolean.TRUE : Boolean.FALSE);
}
```

Static factory methods, less obvious examples

BigInteger.probablePrime(...)
– returns a BigInteger object which, with high probability, represents some prime (with given length in bits)

EnumSet.of(e1, e3); EnumSet.range(e1, e3);
– same arguments, but they return different things (of the same type, i.e. EnumSet)

Google Guava:
MutableGraph<Integer> graph = GraphBuilder.undirected().build();

Why factory methods instead of constructors?

1. They have names so can be easier to use / read.
2. They may return a reference to a readily-built object (no time/space then wasted for yet another object...).
3. They can return an object of a subclass (any subtype of their return type).

Static factory methods, cont’d

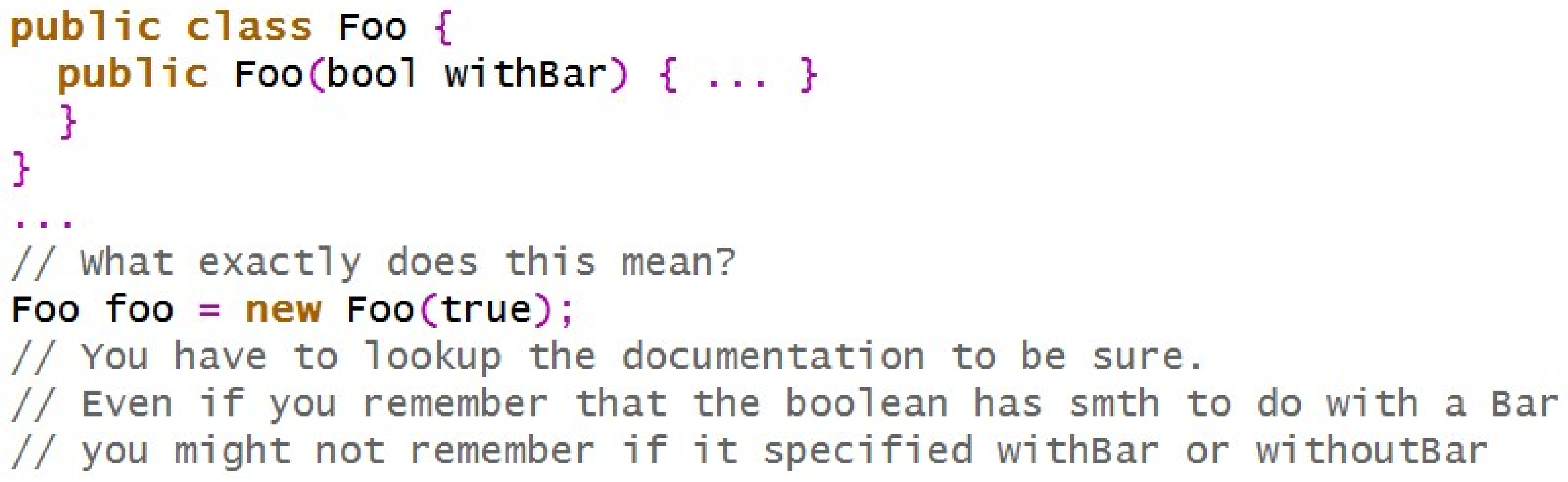
A drawback of static factory methods:
they’re less ‘visible’ in API (javadoc) documentation.

Naming conventions (there are more…):

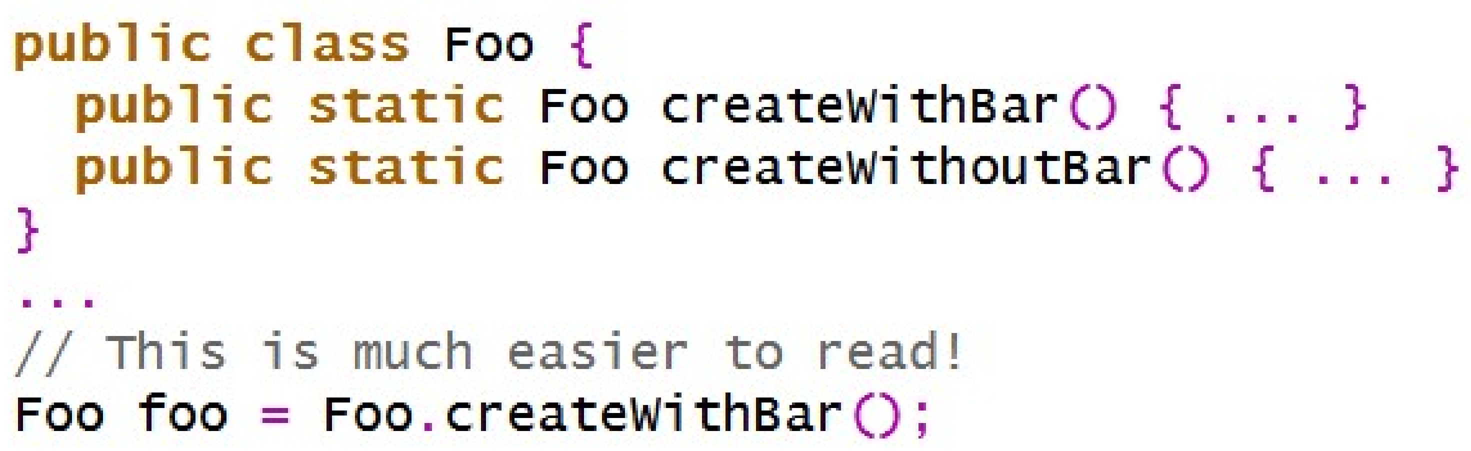
valueOf – mostly for type conversions.
Such a function returns an instance that has, loosely speaking, the same value as its parameters.

getInstance – returns an instance described by its parameters but not having the same value.
E.g., in the case of singleton classes, it returns the sole instance.

Static factory methods are often more readable (than constructors)



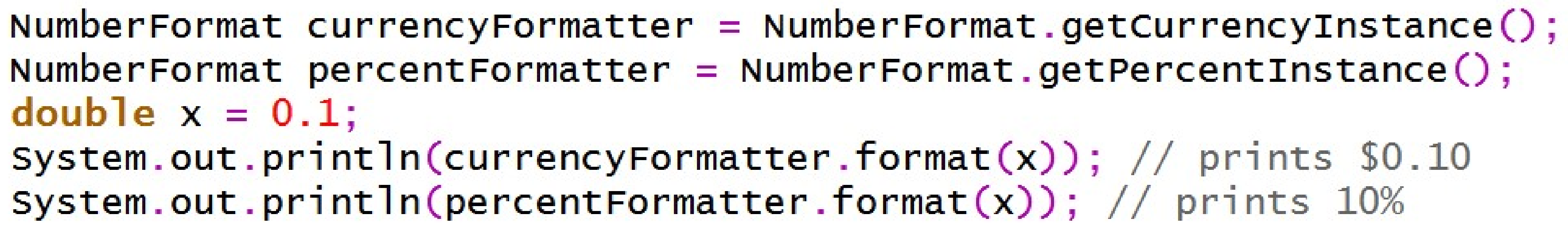
```
public class Foo {
  public Foo(bool withBar) { ... }
  }
}
...
// What exactly does this mean?
Foo foo = new Foo(true);
// You have to lookup the documentation to be sure.
// Even if you remember that the boolean has smth to do with a Bar
// you might not remember if it specified withBar or withoutBar
```



```
public class Foo {
  public static Foo createWithBar() { ... }
  public static Foo createWithoutBar() { ... }
}
...
// This is much easier to read!
Foo foo = Foo.createWithBar();
```

http://stackoverflow.com/questions/929021/what-are-static-factory-methods-in-java

Static factory methods, another example (java.text.NumberFormat)



```
NumberFormat currencyFormatter = NumberFormat.getCurrencyInstance();
NumberFormat percentFormatter = NumberFormat.getPercentInstance();
double x = 0.1;
System.out.println(currencyFormatter.format(x)); // prints $0.10
System.out.println(percentFormatter.format(x)); // prints 10%
```

Two reasons in favor of static factories:

• constructors can’t be given names,

• when you use a constructor, you can’t vary the type of the constructed object. But here: objects of the class DecimalFormat are returned (DecimalFormat extends NumberFormat; actually NumberFormat is an abstract class).

From: Core Java vol. 1, chapter 4

Is returning null bad design?

[based on http://stackoverflow.com/questions/1274792/is-returning-null-bad-design]

In most cases, yes.
E.g., if your filtering method
(with an array/list as the returned value) returns 0 elements,
return an empty array/list rather than null.

Good use of return null

If null is a valid functional result, e.g.
FindFirstObjectThatNeedsProcessing() can return null
if not found and the caller should check accordingly.

EVEN BETTER: return Optional<...> (Java 8+).

Is returning null bad design, cont’d

Bad uses:
trying to replace or hide exceptional situations such as:
- catch(...) and return null,
- API dependency initialization failed,
- invalid input parameters
  (→ inputs must be sanitized by the caller).

Because:
- a null return value provides no meaningful error info,
- the immediate caller often cannot handle the
  error condition,
- the caller may forget to check for nulls.

Exceptions. What if we didn't have them?

[http://www1.cs.columbia.edu/~lok/3101/lectures/04-softwareeng.pdf]

```
errno readFile() {
    int errorCode = 0;
    open the file;
    if (theFileIsOpen) { // determine file length
        if (gotTheFileLength) { // allocate memory;
            if (gotEnoughMemory) { // read the file
                if (readFailed) { errorCode = -1; }
            } else { errorCode = -2; }
        } else { errorCode = -3; }
        close the file;
        if (fileNotClosed && errorCode == 0) {
            errorCode = -4;
        }
        else { errorCode = errorCode and -4; }
    } else { errorCode = -5; }
    return errorCode;
}
```

20

Exception approach

[http://www1.cs.columbia.edu/~lok/3101/lectures/04-softwareeng.pdf]

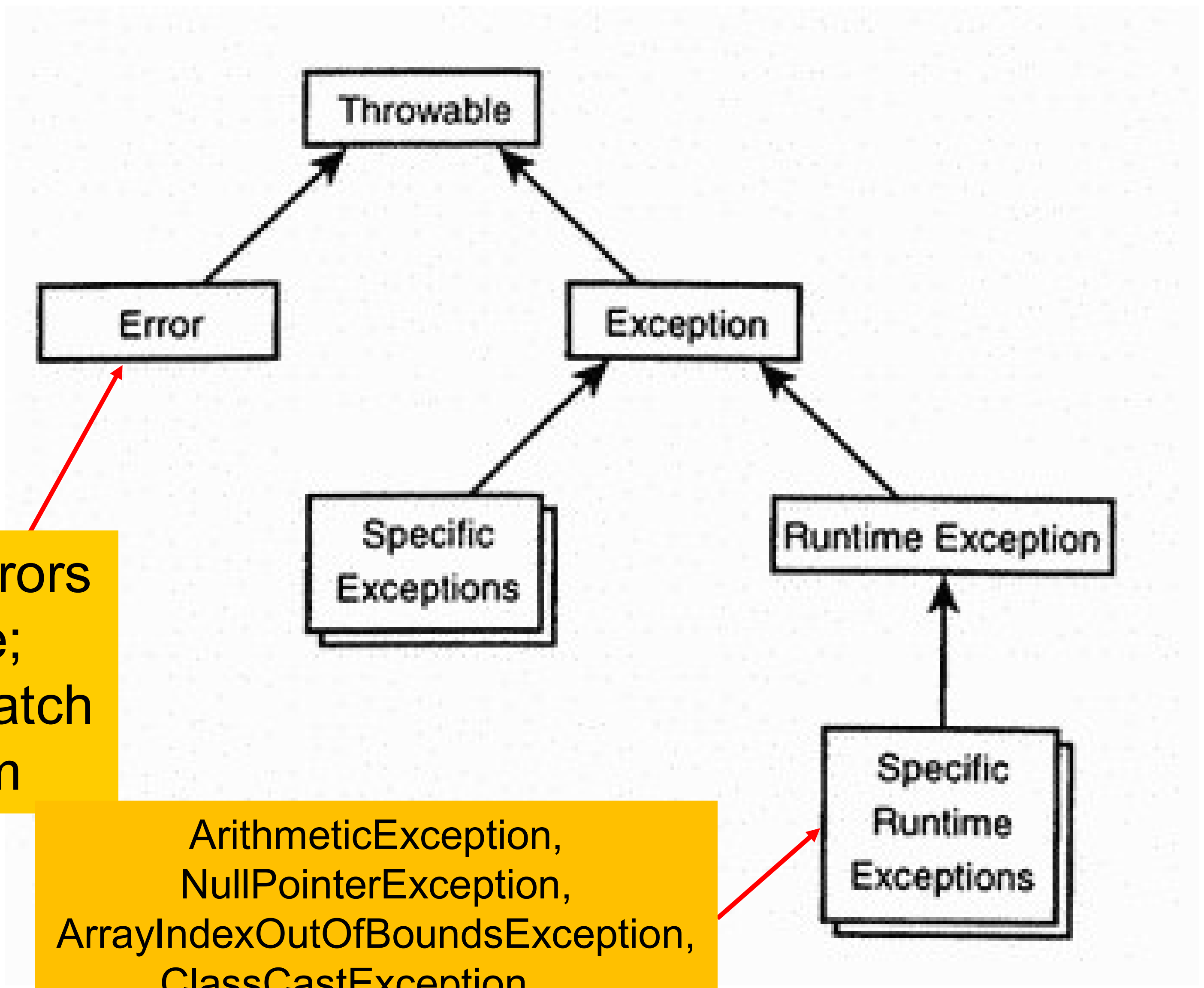
An exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions.

Examples: trying to open a non-existent file, division by zero.

```
readFile() {
  try {
            open the file;
            determine its size;
            allocate that much memory;
            read the file into memory;
            close the file;
  } catch (fileOpenFailed) {
            doSomething;
  } catch (sizeDeterminationFailed) {
            doSomething;
            ..........
  } catch (fileCloseFailed) {
            doSomething;
  }
}
```

21

Exception class hierarchy



fatal errors here; don't catch them

ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException, ClassCastException, …

22

Checked vs. unchecked exceptions

[members.tripod.com/~psdin/comlang/basicJava.PDF]

Unchecked exceptions: RuntimeException, Error, and their subclasses.

A method is NOT oblidged to deal with them.

Checked exceptions: those which are not unchecked. ☺

You HAVE TO catch them: either directly in the method in which the exception occurs, or pass on the method that can throw a checked exception (directly or undirectly), to its caller.

23

Exception stuff keywords

try { ... } – specifies a block of code that might throw an exception

catch (ExceptionClass e) { ... } – exception handler; handles the type of exception indicated by its argument

finally { ... } – this block **always** (*) executes when the *try* block exits (*) except for System.exit(...)

throw someThrowableObject – simply throws a given exception

public void someMethod() throws SomeException { ... } – specifies what someMethod() can throw.
A method can only throw those checked exceptions that are specified in its *throws* clause.

24

How to catch smartly

The order of *catch* blocks DOES matter.

The execution goes (only) to the first catch block that fits the occurring exception. Many might fit!

The conclusion is therefore: first specify more specific exceptions, then more general ones.

E.g.

```
try { ... }
catch (FileNotFoundException e) { ... }  // subclass
catch (IOException e) { .... }  // superclass
/* after try { ... } block (and possibly handling an
exception) we are here */
```

25

finally. Look at this!

```java
public class Ex1 {
  public static void main(String[] args) {
    int[] arr = {2, 5, 1, 4};
    try {
        int i = 0;
        while (true) System.out.println(arr[i++]);
    }
    catch(ArrayIndexOutOfBoundsException e) {
          System.out.println("2 / 0 = " + 2/0);
          System.out.println("I'm in catch(...) block");
    }
    finally {
      System.out.println("I'm in finally{...} block");
    }
    System.out.println("done!");
  }
}
```

Output

```
2
5
1
4
I'm in finally{...} block
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Ex1.main(Ex1.java:11)
```

26

finally. Modified code

```java
public class Ex1 {
  public static void main(String[] args) {
    int[] arr = {2, 5, 1, 4};
    try {
        int i = 0;
        while (true) System.out.println(arr[i++]);
    }
    catch(ArrayIndexOutOfBoundsException e) {
       try {
          System.out.println("2 / 0 = " + 2/0);
          System.out.println("I'm in catch(...) block");
       }
       catch(ArithmeticException e1) {
         System.out.println("Wanna divide by zero?!");
       }
       finally{
         System.out.println("I'm in catch(e1) / finally block");
       }
    }
    finally {
      System.out.println("I'm in finally{...} block");
    }
    System.out.println("done!");
  }
}
```

output

```
2
5
1
4
Wanna divide by zero?!
I'm in catch(e1) / finally block
I'm in finally{...} block
done!
```

27

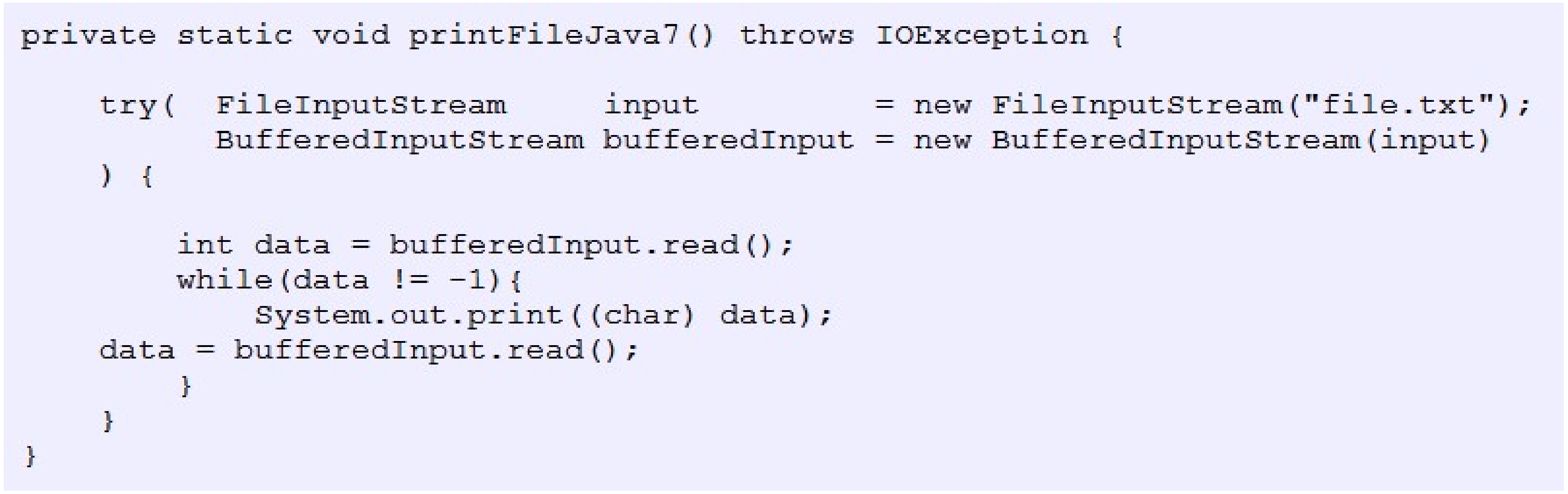
try-with-resources

```
private static void printFileJava7() throws IOException {

    try(FileInputStream input = new FileInputStream("file.txt")) {

        int data = input.read();
        while(data != -1){
            System.out.print((char) data);
            data = input.read();
        }
    }
}
```

When the try block finishes the FileInputStream
will be closed automatically.
This is possible because FileInputStream implements
the interface java.lang.AutoCloseable.
All classes implementing this interface
can be used inside the try-with-resources construct.

http://tutorials.jenkov.com/java-exception-handling/try-with-resources.html

Using multiple resources



```
private static void printFileJava7() throws IOException {

    try(  FileInputStream     input         = new FileInputStream("file.txt");
          BufferedInputStream bufferedInput = new BufferedInputStream(input)
    ) {

        int data = bufferedInput.read();
        while(data != -1){
            System.out.print((char) data);
    data = bufferedInput.read();
        }
    }
}
```

The resources will be closed in reverse order of the order in which they are created / listed inside the parentheses. First BufferedInputStream will be closed, then FileInputStream.

Recommended read:
https://github.com/google/guava/wiki/ClosingResourcesExplained
Why old-style (with try-catch-finally finally) is full of issues.

http://tutorials.jenkov.com/java-exception-handling/try-with-resources.html

Try-with-resources in Java 9

[https://blogs.oracle.com/darcy/entry/concise_twr_jdk9]

For example, given resource declarations like

```
 final Resource resource1 = new Resource("resource1"); // final resource
 Resource resource2 = new Resource("resource2"); // effectively final resource
```

the old way to write the code to manager these resources would be e.g.:

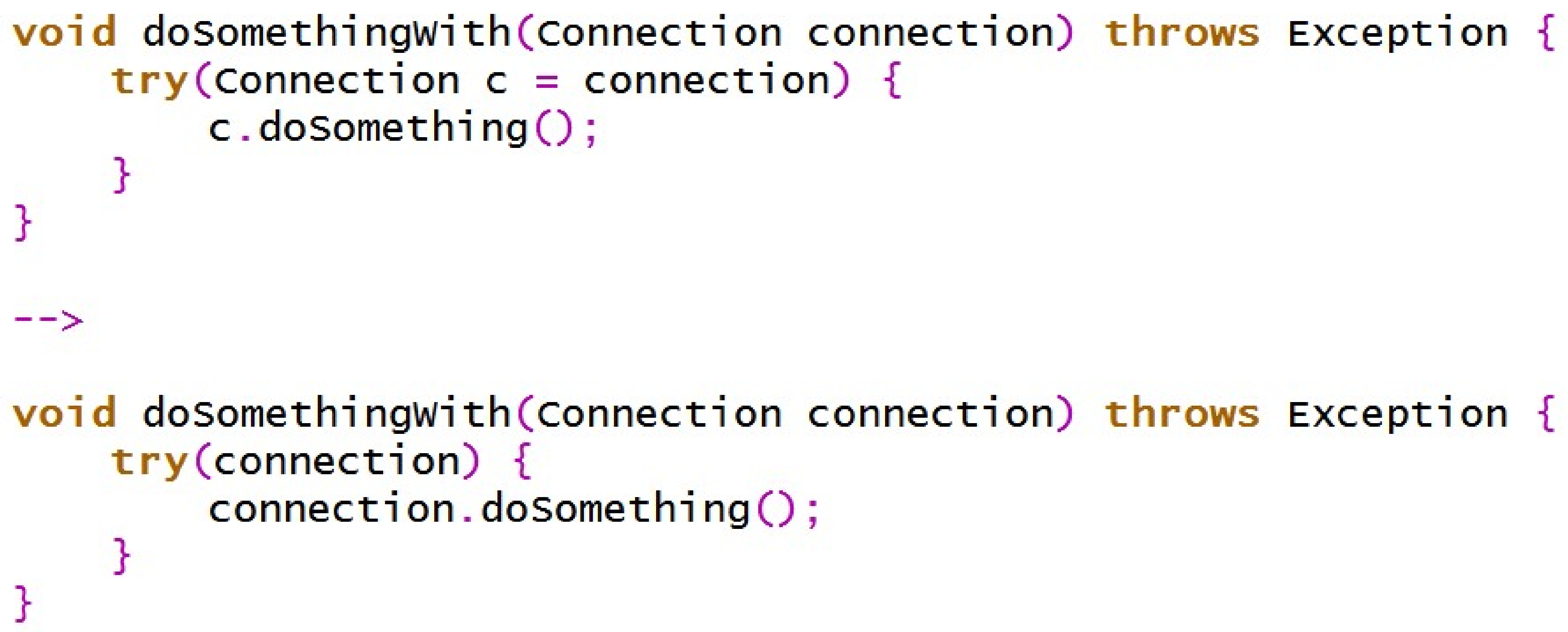
```
 // Original try-with-resources statement from JDK 7 or 8
 try (Resource r1 = resource1;
    Resource r2 = resource2) {
    // Use of resource1 and resource2 through r1 and r2.
 }
```

while the new way can be just

```
// New and improved try-with-resources in JDK 9
try (resource1;
      resource2) {
     // Use of resource1 and resource2.
}
```

Try-with-resources in Java 9, example

[https://www.sitepoint.com/ultimate-guide-to-java-9/]

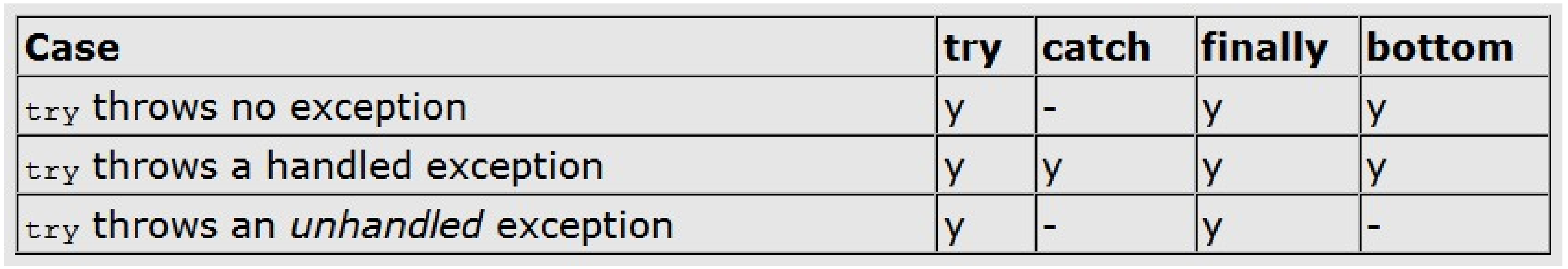


```
void doSomethingWith(Connection connection) throws Exception {
    try(Connection c = connection) {
        c.doSomething();
    }
}

-->

void doSomethingWith(Connection connection) throws Exception {
    try(connection) {
        connection.doSomething();
    }
}
```

Control flow with exceptions, summary

If an exception occurs outside a try...catch block, the exception simply propagates "up" to the caller.

If an exception occurs inside a try…catch block, then the control flow can take different forms. Here are the cases, along with indicators for which code blocks are included in the control flow ("bottom" refers simply to any code which follows the finally block):

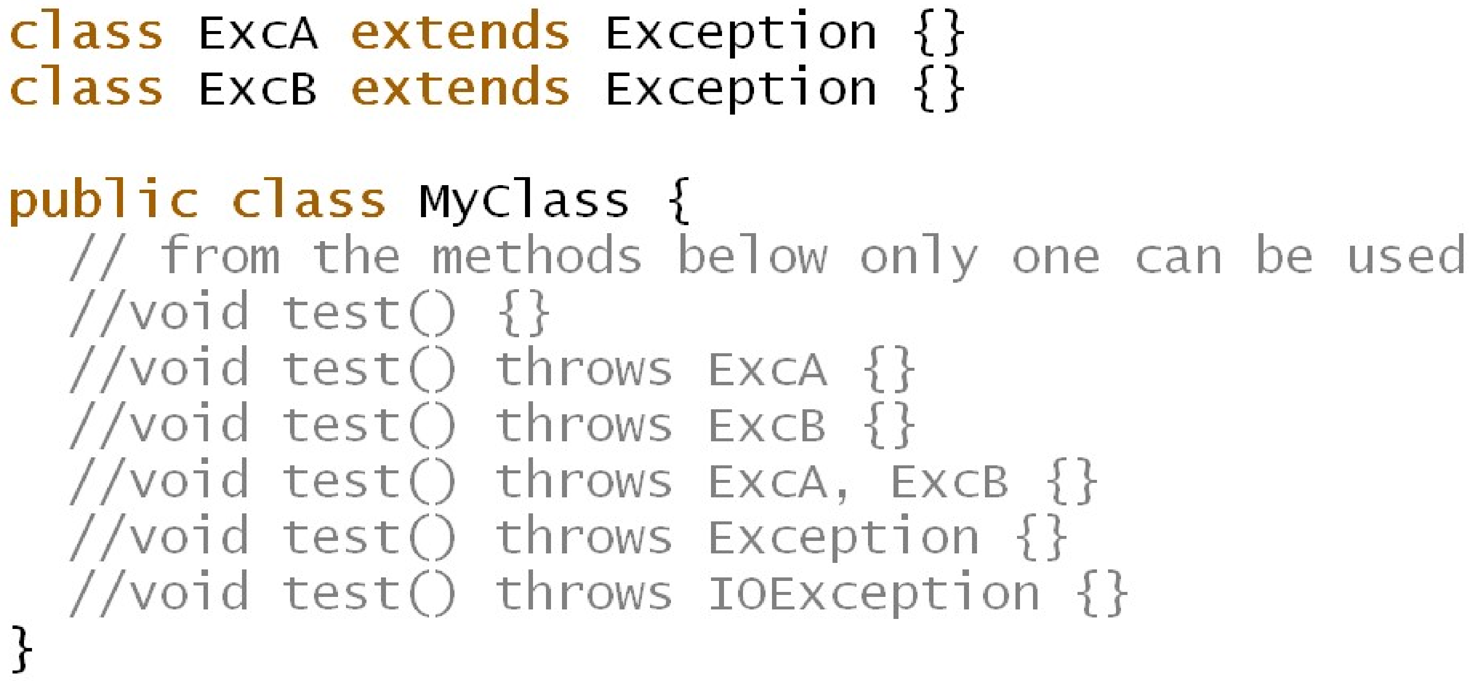


Case	**try**	**catch**	**finally**	**bottom**
`try` throws no exception	y	-	y	y
`try` throws a handled exception	y	y	y	y
`try` throws an *unhandled* exception	y	-	y	-

http://www.javapractices.com/topic/TopicAction.do;jsessionid=58024221061E46C20E9B231A3439ED0C?Id=19

Exception declaration IS NOT part of the signature

[based on
http://wazniak.mimuw.edu.pl/index.php?title=PO_Wyj%C4%85tki__%C4%87wiczenia]



```
class ExcA extends Exception {}
class ExcB extends Exception {}

public class MyClass {
  // from the methods below only one can be used
  //void test() {}
  //void test() throws ExcA {}
  //void test() throws ExcB {}
  //void test() throws ExcA, ExcB {}
  //void test() throws Exception {}
  //void test() throws IOException {}
}
```

Recall: similarly the return value is not part of the method signature.

Exceptions: some best practices

Use meaningful messages.

Not:

```
if (list.size() != array.length)
    throw new IllegalArgumentException("Wrong data");
```

But rather:

```
if (list.size() != array.length)
    throw new IllegalArgumentException("List and array have
         different sizes: " + list.size() + " " + array.length);
```

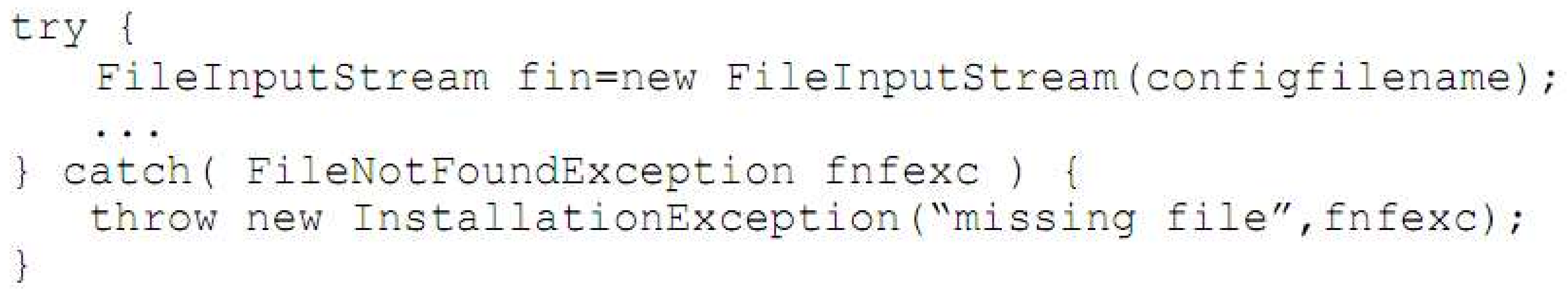
Sometimes we catch a standard exception and wrap it into a custom one.

```
public void wrapException(String input) throws MyBusinessException {
    try {
        // do something
    } catch (NumberFormatException e) {
        throw new MyBusinessException("Msg to describe the error.", e);
    }
}
```

Use of unchecked exceptions

Program bugs (e.g. signalled with IndexOutOfBoundsException) should be fixed, not treated in code. Yet…

```
public class ProgrammingException extends RuntimeException
 try {
   Date defaultDate=format.parse(DEFAULT_DATE_STRING);
   ...
 } catch(ParseException pexc) {
     // If this exception is thrown, I got something wrong
     throw new ProgrammingException("bad init value", pexc);
     // exception chaining is used, so that the info associated with
     // the original ParseException is NOT lost
 }
```



```
try {
    FileInputStream fin=new FileInputStream(configfilename);
    ...
} catch( FileNotFoundException fnfexc ) {
    throw new InstallationException("missing file",fnfexc);
}
```

A puzzle: how to draw a random int from [0, n)?

[based on J. Bloch, *Effective Java*, 2001, Chap. 7]

```
1. Random rnd = new Random();  // java util.Random
    ...
   r = Math.abs(rnd.nextInt()) % n;

2. r = (int)(Math.random() * n);

3. r = rnd.nextInt(n);  // after Random rnd = …
```

Ad 1: consider
n = 2 * (Integer.MAX_VALUE / 3);

?!?!

Assertions

A mechanism known from many languages
(C/C++, Ada, Python...)
to test a given assumption about your program.
Useful in debugging.

E.g. water in liquid state in ambient conditions
has temperature between 0 and 100 (degree Celsius) –
if it is -20 or 356, smth is wrong with your code...

But you can set a line in Java (since v1.4):

```
assert temp >= 0 && temp <= 100 : "Weird water!";
```

Optional expression. Its value is passed to an appropriate AssertionError constructor.

By default, assertions are disabled

Enable assertions at runtime:
java -ea AssertTest
(or -enableassertions instead of -ea)

Assertions can also be turned on/off selectively,
in pointed classes or packages (for details, see
http://docs.oracle.com/javase/8/docs/technotes/guides/language/assert
.html).

Some people advise to keep
assertions enabled even in the final (production) build.
Esp. in non-performance-critical parts.
That’s because bugs may pop up
months or years after product release.
Assertions make reporting bugs much easier.

When not to use assertions

[http://download.oracle.com/javase/1.4.2/docs/guide/lang/assert.html]

good code

```
/**
 * Sets the refresh rate.
 *
 * @param  rate refresh rate, in frames per second.
 * @throws IllegalArgumentException if rate <= 0 or
 *           rate > MAX_REFRESH_RATE.
 */
 public void setRefreshRate(int rate) {
     // Enforce specified precondition in public method
     if (rate <= 0 || rate > MAX_REFRESH_RATE)
         throw new IllegalArgumentException("Illegal rate: " + rate);

     setRefreshInterval(1000/rate);
 }
```

Do not use assertions to check the parameters of a public method!

1. If some precondition exists (like in the above example), the params should always be checked (and assertions may be off).
2. Specific exception type should be used, not AssertionError (e.g., IllegalArgumentException, ArrayOutOfBoundsException).

Assert use cases

(from http://www.javapractices.com)

- post-conditions – verify the promises
  made by a method to its caller,
- class invariants – validate object state,
- pre-conditions (in private methods only),
- unreachable-at-runtime code – parts of the program
  which you expect to be unreachable,
  but which cannot be verified as such at compile-time
  (often else clauses and default cases in switch state)

Checking function parameters’ validity

[J. Bloch, *Effective Java*, 3rd Ed., Item 49]

Especially important to check parameters
that are not used by a method,
but stored for later use.

Example: a static factory method, which takes an int array and returns a List view of the array.

```
static List<Integer> intArrayAsList(int[] a) {
  ...
  return new AbstractList<>() {
    ...  // get, set, size methods overridden
  };
}
```

Imagine that you pass a null to intArrayAsList…

Checking function parameters' validity, cont'd

[J. Bloch, *Effective Java*, 3rd Ed., Item 49]

With a check (e.g., with Objects.requireNonNull): the method would throw a NullPointerException (fail fast, that's good!).

Without a check: the method would return a reference to a newly created List instance that would throw a NullPointerException only when a client attempted to use it (hard to track the origin of the error!).

Inner classes

[http://docs.oracle.com/javase/tutorial/java/javaOO/innerclasses.html]

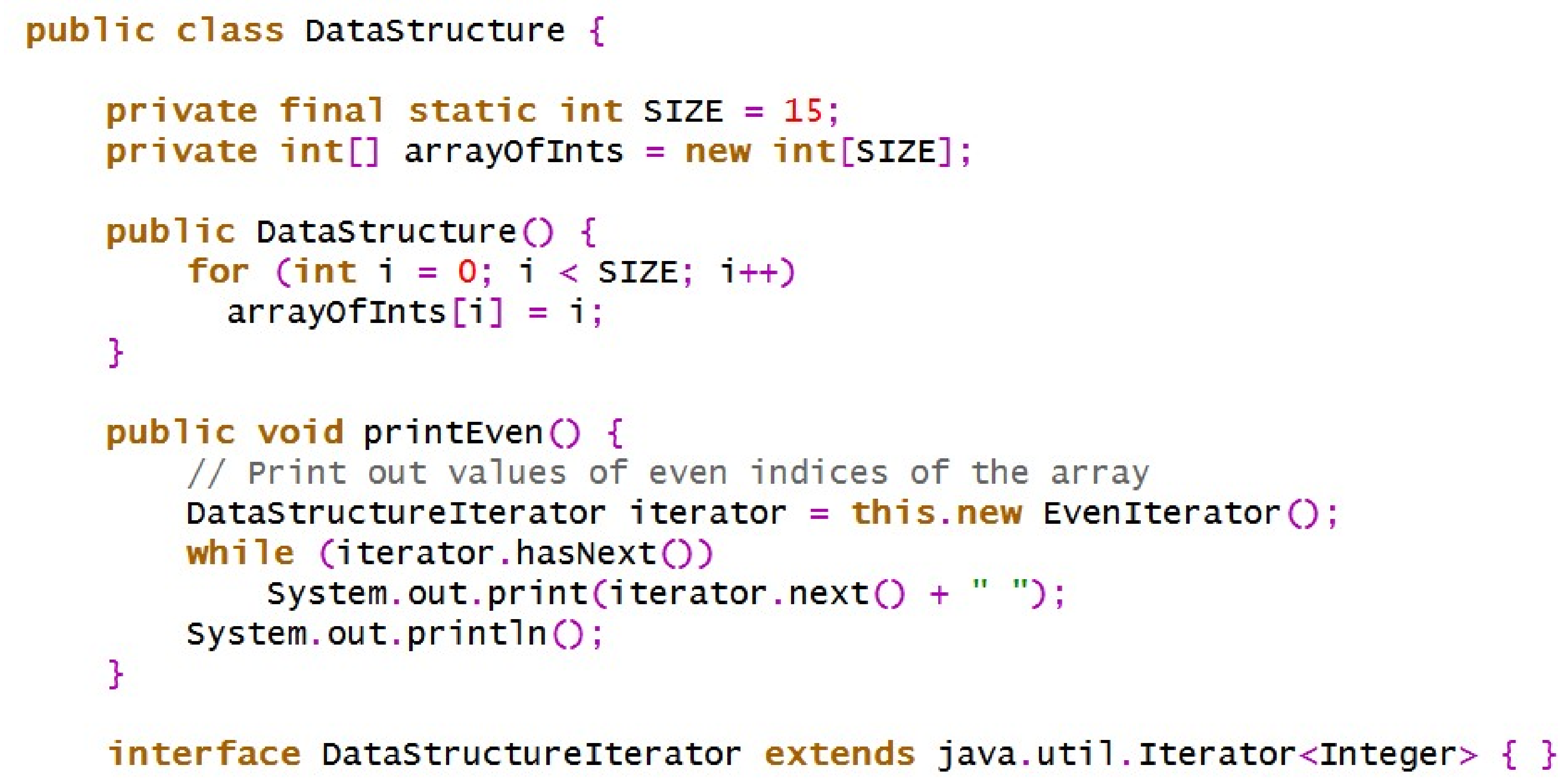
An inner class is a class declared inside another class.
You can declare a class inside another,
much like you declare fields and methods.
The class that encloses the declaration of the inner class
is called the outer class.

Inner classes are used primarily to implement
helper classes.
You can declare an inner class within the body of a
method (this is called a local inner class).
Can also declare an inner class within the body of a
method without naming it (anonymous inner classes).

43

Inner class example (1 / 2)

[http://docs.oracle.com/javase/tutorial/java/javaOO/innerclasses.html]



```
public class DataStructure {

    private final static int SIZE = 15;
    private int[] arrayOfInts = new int[SIZE];

    public DataStructure() {
        for (int i = 0; i < SIZE; i++)
          arrayOfInts[i] = i;
    }

    public void printEven() {
        // Print out values of even indices of the array
        DataStructureIterator iterator = this.new EvenIterator();
        while (iterator.hasNext())
            System.out.print(iterator.next() + " ");
        System.out.println();
    }

    interface DataStructureIterator extends java.util.Iterator<Integer> { }
```

44

Inner class example (2 / 2)

```
    // Inner class implements the DataStructureIterator interface,
    // which extends the Iterator<Integer> interface

    private class EvenIterator implements DataStructureIterator {

        // Start stepping through the array from the beginning
        private int nextIndex = 0;

        public boolean hasNext() {
            // Check if the current element is the last in the array
            return (nextIndex <= SIZE - 1);
        }

        public Integer next() {
            // Record a value of an even index of the array
            Integer retValue = Integer.valueOf(arrayOfInts[nextIndex]);

            nextIndex += 2;  // Get the next even element
            return retValue;
        }
    }

    public static void main(String s[]) {
        DataStructure ds = new DataStructure();
        ds.printEven(); // Print out only values of even indices
    }
}
```

Output: 0 2 4 6 8 10 12 14

Closing an AWT window (and more...)

```
import java.awt.*;
import java.awt.event.*;

public class HelloApp extends Frame implements WindowListener
{
   public HelloApp()
   {
      super();
      addWindowListener(this);
      setSize(320, 200);
      setTitle("Simple Java App");
      setVisible(true);
   }

   public static void main(String args[])
   {  new HelloApp();  }

   public void windowClosing(WindowEvent e) { System.exit(0); }
   public void windowClosed(WindowEvent e) { }
   public void windowOpened(WindowEvent e) { }
   public void windowIconified(WindowEvent e) { }
   public void windowDeiconified(WindowEvent e) { }
   public void windowActivated(WindowEvent e) { }
   public void windowDeactivated(WindowEvent e) { }
}
```

All that is pretty boring, isn't it?

46

Using adapters

Implementing WindowListener, MouseListener etc. inferfaces required in human-interactive applications may be tedious. We often don’t need to implement all the required methods.
Hence – null (empty, dummy) methods.

A solution we know is to create an adapter
(with dummy methods only)
and extend it in our class.
This is often unacceptable since then
we cannot extend any other class!

Anything better?

47

Anonymous classes

[http://cs.nyu.edu/~yap/classes/visual/03s/lect/l7/prog/EmptyFrame1.java]

Java provides the class WindowAdapter (java.awt.event.*) which implements WindowListener with all 7 methods defaulted to the null implementation. But extending WindowAdapter still doesn't solve the problem (of course). So we are going to implement an anonymous extension of WindowAdapter().

```
....
addWindowListener (
    new WindowAdapter()
    {
            public void windowClosing(WindowEvent e)
            { System.exit(0); }
    }
);
```

The parameter for method addWindowListener(…) is an object of a class implementing WindowAdapter.

Anonymous classes, more notes

[http://mindprod.com/jgloss/anonymousclasses.html]

You define it, and create an object of that type as a parameter all in one line. In other words, an anonymous class is defined by a Java expression, not a Java statement.

Syntax a bit strange: no *class* or any other keyword (e.g. *extends*).

You can refer to **this** of the outer class via **MyOuterClass.this**. You can refer to the outer class's methods by myOuterInstanceMethod() or MyOuterClass.this.myOuterInstanceMethod().

Objects of anonymous classes have access to **final** local variables that are declared within the scope of the anonymous class.

Unlike a local class,
an anonymous class cannot have a constructor.

49

Java Collection Framework, basics

A collections framework is a unified architecture
for representing and manipulating collections,
allowing them to be manipulated
independently of the details of their representation.

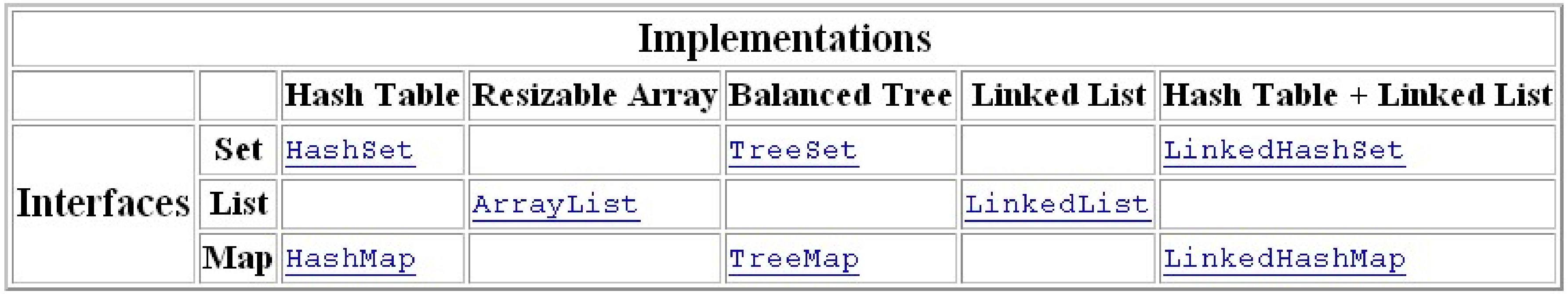
Why using JCF?
• less programming effort
(useful data structures and algorithms provided),
• good performance (often not that easy to implement so
efficient and robust structures / algs on one's own),
• easier to understand code (esp. someone else's).

JCF components: interfaces (such as sets, lists and maps),
implementations, algorithms (static methods to sort a
collection, reverse it etc.), iterators.

50

Basic collections

- lists: sequential or random access; duplicates allowed
- sets: no indexed access, duplicates forbidden
- maps aka dictionaries, associative arrays



		Implementations				
		Hash Table	Resizable Array	Balanced Tree	Linked List	Hash Table + Linked List
Interfaces	Set	HashSet		TreeSet		LinkedHashSet
	List		ArrayList		LinkedList	
	Map	HashMap		TreeMap		LinkedHashMap

Legacy collections: Vector and Hashtable.

51

Generic types

Prefer generic to raw types.
E.g., use List<Integer> rather than List.

Why not raw types:
- casting needed when taking an element out of a collection,
- cast can fail at runtime (→ unsafe),
- more cumbersome usage with lambdas (next lecture).

Let’s start from an array...

Pros:

- knows the type it holds, i.e. compile-type checking
- knows its size
- can hold primitive types directly (=no wrapping).

Cons:

- fixed size (=static)
- can hold only one type of objects.

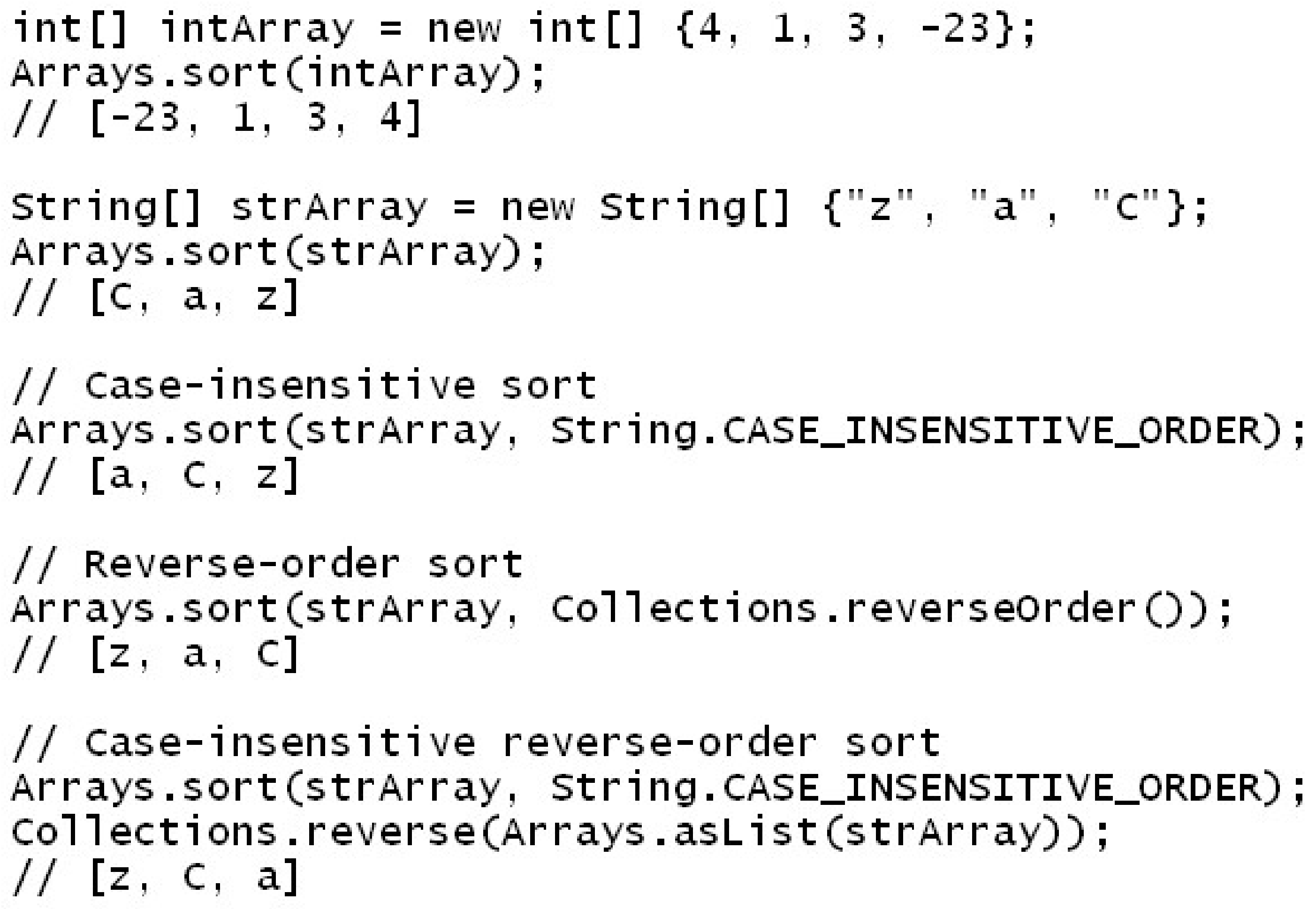
Helper class: java.utils.Arrays with basic functionalities (public static methods):

- equals()
- fill() // instantiation
- asList() // conversion
- binarySearch(), sort()

53

Sorting an array

[http://javaalmanac.com/egs/java.util/coll_SortArray.html?l=rel]



```
int[] intArray = new int[] {4, 1, 3, -23};
Arrays.sort(intArray);
// [-23, 1, 3, 4]

String[] strArray = new String[] {"z", "a", "C"};
Arrays.sort(strArray);
// [C, a, z]

// Case-insensitive sort
Arrays.sort(strArray, String.CASE_INSENSITIVE_ORDER);
// [a, C, z]

// Reverse-order sort
Arrays.sort(strArray, Collections.reverseOrder());
// [z, a, C]

// Case-insensitive reverse-order sort
Arrays.sort(strArray, String.CASE_INSENSITIVE_ORDER);
Collections.reverse(Arrays.asList(strArray));
// [z, C, a]
```

54

Arrays.asList(…)

```
Card[] cardDeck = new Card[52];
. . .
List<Card> cardList = Arrays.asList(cardDeck);
```

The returned object is NOT an ArrayList!
It's a view object.

All methods that would change the array size
(e.g., add) throw an UnsupportedOperationException.

Views are used also in other classes, e.g.
Collections.nCopies(…),
List<E> subList(int fromIndex, int toIndex) (in the interface List<E>),
SortedMap<K, V> headMap(K to) (in the interface SortedMap<K, V>).

Does it work?

```
Point[] a = new Point[10];
Object[] b;
b = a;
b[0] = new Point(10,20);
```

a) doesn’t compile,
b) compiles, but throws ArrayStoreException
c) compiles, but throws CastClassException
[✓ d) works correctly ?]

```
Point[] a = new Point[10];
Object[] b;
b = a;
b[0] = "hi there";
```

a) doesn’t compile,
[✓ b) compiles, but throws ArrayStoreException]
c) compiles, but throws CastClassException
d) works correctly ?

```
Point[] a = new Point[10];
Object[] b;
b = a;
a[0] = "hi there";
```

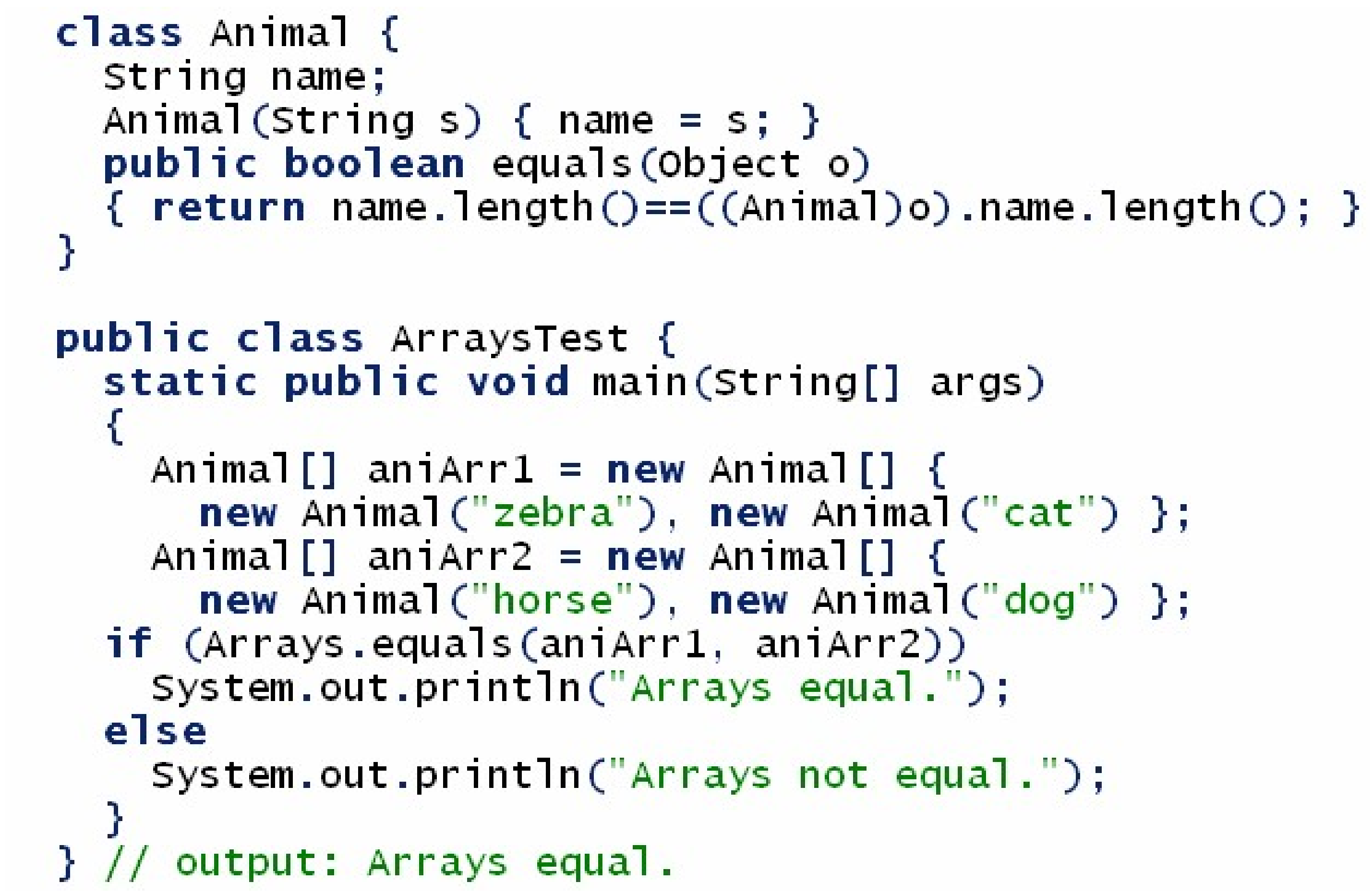
a) doesn’t compile,
[✓ b) compiles, but throws ArrayStoreException]
c) compiles, but throws CastClassException
d) works correctly ?

A similar one..?!

```
jshell> Object[] tab = {"a", "bc"}
tab ==> Object[2] { "a", "bc" }

jshell> tab.getClass()
$13 ==> class [Ljava.lang.Object;

jshell> Object[] tab2 = new String[]{"a", "bc"}
tab2 ==> String[2] { "a", "bc" }

jshell> tab2.getClass()
$23 ==> class [Ljava.lang.String;

jshell> tab[0] = new java.util.Date()
$27 ==> Thu Oct 25 11:27:29 CEST 2018

jshell> tab
tab ==> Object[2] { Thu Oct 25 11:27:29 CEST 2018, "bc" }

jshell> tab2[0] = new java.util.Date()
|  java.lang.ArrayStoreException thrown: java.util.Date
|        at (#30:1)
```

java.util.Arrays methods (1/10)

equals – overloaded for each primitive type plus one more for `Object` arrays: for the case of `Object` arrays, two objects `o1`, `o2` are equal if either both are `null`, or `o1.equals(o2)`.



```java
class Animal {
  String name;
  Animal(String s) { name = s; }
  public boolean equals(Object o)
  { return name.length()==((Animal)o).name.length(); }
}

public class ArraysTest {
  static public void main(String[] args)
  {
    Animal[] aniArr1 = new Animal[] {
      new Animal("zebra"), new Animal("cat") };
    Animal[] aniArr2 = new Animal[] {
      new Animal("horse"), new Animal("dog") };
  if (Arrays.equals(aniArr1, aniArr2))
    System.out.println("Arrays equal.");
  else
    System.out.println("Arrays not equal.");
  }
} // output: Arrays equal.
```

58

java.util.Arrays methods (2/10)

toString – supports printing arrays.

```
String[] arr = new String[] {"cat", "boa constrictor" };
System.out.println(Arrays.toString(arr)); // [cat, boa constrictor]
```

For multi-dim arrays, use deepToString().

```
String[][] arr = new String[][] {  { "a", "1" }, { "b", "2" }, {"c", "3" }  };
System.out.println(Arrays.deepToString(arr)); // [[a, 1], [b, 2], [c, 3]]
```

Prev. example: what if we forget and use toString()?

Output: [[Ljava.lang.String;@1f6a7b9, [Ljava.lang.String;@7d772e, [Ljava.lang.String;@11b86e7]

An array of objects (not primitives):

```
System.out.println(Arrays.asList(strArray)); // e.g. [C, a, z]
```

59

java.util.Arrays methods (3/10)

sort – overloaded for each primitive type (except boolean), plus one more for `Object`.

The array will be sorted in ascending order.

void sort(type[] a) // uses quick sort for primitive types,
// *O*(*n* log *n*) avg time
// acc. to numerical order

void sort(Object[] a) // uses merge sort for object types
// stable and *O*(*n* log *n*) worst case time
// acc. to natural ordering of its elements
// all the array elements must implement
// the Comparable interface

60

java.util.Arrays methods (4/10), sort example

[http://www.onjava.com/pub/a/onjava/2003/03/12/java_comp.html]

```
Person[] persons = new Person[4];
persons[0] = new Person();
persons[0].setFirstName("Elvis");
persons[0].setLastName("Goodyear");
persons[0].setAge(56);

persons[1] = new Person();
persons[1].setFirstName("Stanley");
persons[1].setLastName("Clark");
persons[1].setAge(8);

persons[2] = new Person();
persons[2].setFirstName("Jane");
persons[2].setLastName("Graff");
persons[2].setAge(16);

persons[3] = new Person();
persons[3].setFirstName("Nancy");
persons[3].setLastName("Goodyear");
persons[3].setAge(69);
```

Now, invoke
Arrays.sort(persons); ?

No; it would throw
ClassCastException. ☹

61

java.util.Arrays methods (5/10), sort example, cont'd

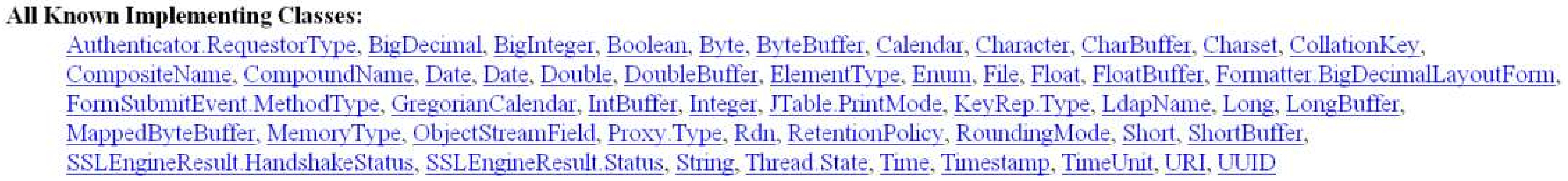
[http://www.onjava.com/pub/a/onjava/2003/03/12/java_comp.html]

Solution: of course we have to implement *Comparable* interface.

It has one method, *compareTo*, which determines how to compare two instances of the class.

public int compareTo(Object o)

/* returns 0 if equal, a negative integer if the curr object is smaller than *o*, and a positive integer otherwise */



All Known Implementing Classes:

Authenticator.RequestorType, BigDecimal, BigInteger, Boolean, Byte, ByteBuffer, Calendar, Character, CharBuffer, Charset, CollationKey, CompositeName, CompoundName, Date, Date, Double, DoubleBuffer, ElementType, Enum, File, Float, FloatBuffer, Formatter.BigDecimalLayoutForm, FormSubmitEvent.MethodType, GregorianCalendar, IntBuffer, Integer, JTable.PrintMode, KeyRep.Type, LdapName, Long, LongBuffer, MappedByteBuffer, MemoryType, ObjectStreamField, Proxy.Type, Rdn, RetentionPolicy, RoundingMode, Short, ShortBuffer, SSLEngineResult.HandshakeStatus, SSLEngineResult.Status, String, Thread.State, Time, Timestamp, TimeUnit, URI, UUID

62

java.util.Arrays methods (6/10), sort example, cont’d

[http://www.onjava.com/pub/a/onjava/2003/03/12/java_comp.html]

```
class Person implements Comparable {
  private String firstName;
  private String lastName;
  private int age;
```

instanceof

```
public int compareTo(Object anotherPerson) throws ClassCastException {
  if (!(anotherPerson instanceof Person))
    throw new ClassCastException("A Person object expected.");
  int anotherPersonAge = ((Person) anotherPerson).getAge();
  return this.age - anotherPersonAge;
}
```

Now, in some other class (e.g. Testing)
we create several Person objects, they are stored in e.g.
array *persons*, and sort them with Arrays.sort(persons).

63

java.util.Arrays methods (7/10), using java.util.Comparator

[http://www.onjava.com/pub/a/onjava/2003/03/12/java_comp.html]

Ok, we can sort objects using *Comparable*, why bothering with another interface (*Comparator*)?

That's because objects sometimes need to be compared in different ways (e.g. people – by their last names in lex. order, or by birth date, etc.).
Idea: pass an external object to method sort(...).
An object of a class implementing Comparator interface.

java.util.Comparator interface has one method:
public int compare(Object o1, Object o2)
// o1 less than o2 – return a negative int etc.

64

Arrays methods (8/10), Comparator example, cont’d

```java
package comparable.ex02;

import java.util.Comparator;

public class LastNameComparator implements Comparator {
  public int compare(Object person, Object anotherPerson) {
    String lastName1 = ((Person) person).getLastName().toUpperCase();
    String firstName1 = ((Person) person).getFirstName().toUpperCase();
    String lastName2 = ((Person) anotherPerson).getLastName().toUpperCase();
    String firstName2 = ((Person) anotherPerson).getFirstName().toUpperCase();

    if (!(lastName1.equals(lastName2)))
      return lastName1.compareTo(lastName2);
    else
      return firstName1.compareTo(firstName2);
  }
}
```

```java
public class FirstNameComparator implements Comparator {
  public int compare(Object person, Object anotherPerson) {
    String lastName1 = ((Person) person).getLastName().toUpperCase();
    String firstName1 = ((Person) person).getFirstName().toUpperCase();
    String lastName2 = ((Person) anotherPerson).getLastName().toUpperCase();
    String firstName2 = ((Person) anotherPerson).getFirstName().toUpperCase();
    if (!(firstName1.equals(firstName2)))
      return firstName1.compareTo(firstName2);
    else
      return lastName1.compareTo(lastName2);
  }
}
```

java.util.Arrays methods (9/10), using java.util.Comparator, example, cont'd

[http://www.onjava.com/pub/a/onjava/2003/03/12/java_comp.html]

Testing the Comparator objects:

create some extra class (e.g. Testing), create an array *persons* in it, fill it with data, then call

Arrays.sort(persons, new FirstNameComparator());

or

Arrays.sort(persons, new LastNameComparator());

Problem is, if we need *n* comparison criteria, we also need to create *n* extra classes. Lots of clutter. This can be mitigated by making the comparators anonymous classes, or even better with lambda expressions in Java 8.

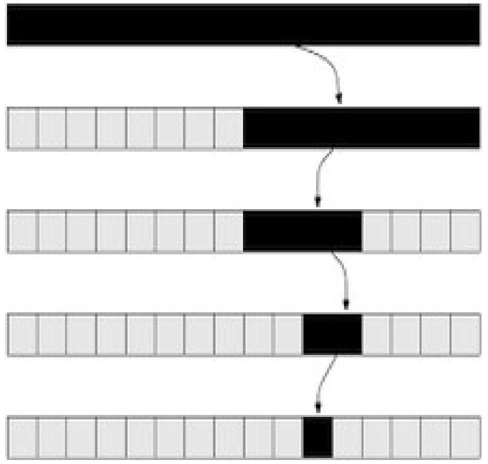
66

java.util.Arrays methods (10/10)

binarySearch – overloaded for each primitive type (except boolean), plus one more for `Object`.
The array must be sorted in ascending order.

int binarySearch (type[] array, type key)
int binarySearch (Object[] array, Object key, Comparator comp)

It returns the index to the position where the key is found (if duplicates exist, we cannot be sure about the exact position!), or if it is not found, to the insertion point.
(If *key* greater than all the items, the method returns *array.length*.)



Binary search in a sorted array of size *n*: $O(\log n)$ worst-case time.

Don’t you ever binary search in an unsorted array!
(Result would be unpredictable.)

67

Lambda expression for a comparator (Java 8)

```
String[] a = "This is a text example".split("\\s");
Arrays.sort(a, (s1, s2) -> s1.compareToIgnoreCase(s2));
System.out.println(Arrays.asList(a));

// [a, example, is, text, This]
```

List interface

List – an ordered collection (*sequence*).
Precise control over where in the list
each element is inserted.
The user can access elements by their index
(position in the list, starting from 0),
and search for elements in the list.

Access by index: no constant-time guarantee
(for an arbitrary list),
hence use iterating rather than indexing through a List!
Use Iterator (forward traversal only) or ListIterator
(bidirectional).

There's a useful marker interface RandomAccess.
For a given List li, use: if (li instanceof RandomAccess) …

69

List interface, cont’d

Selected methods (“*opt*” – optional):

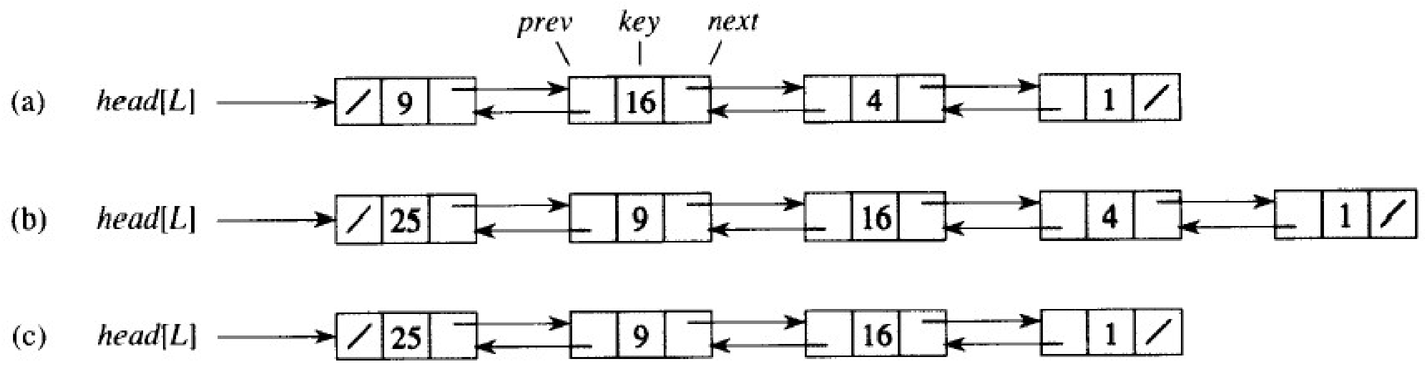
opt	void	add(int index, Object element)	
opt	boolean	add(Object o)	// false if the Object *o* was not added
opt	boolean	addAll(Collection c)	// true if the list changed as a result
opt	void	clear()	
	boolean	contains(Object o)	
	Object	get(int index)	
	int	indexOf(Object o)	// -1 if the list doesn’t contain element *o*
	boolean	isEmpty()	
	int	lastIndexOf(Object o)	
opt	Object	remove(int index)	// Returns the element removed
opt	Object	set(int index, Object element)	// replaces an item at *index*
	int	size()	

70

List implementations, short comparison

ArrayList – internally an array, so it has *O*(1) access to an indexed item, but slow (*O*(*n*)) update (insert / delete) from the middle of the structure.

LinkedList – fast (constant-time) update in a pointed position, slow access (*O*(*n*) – needs traversal over the list).



From [Cormen, McGraw-Hill, 2000, p. 205]

Doubly-linked list.

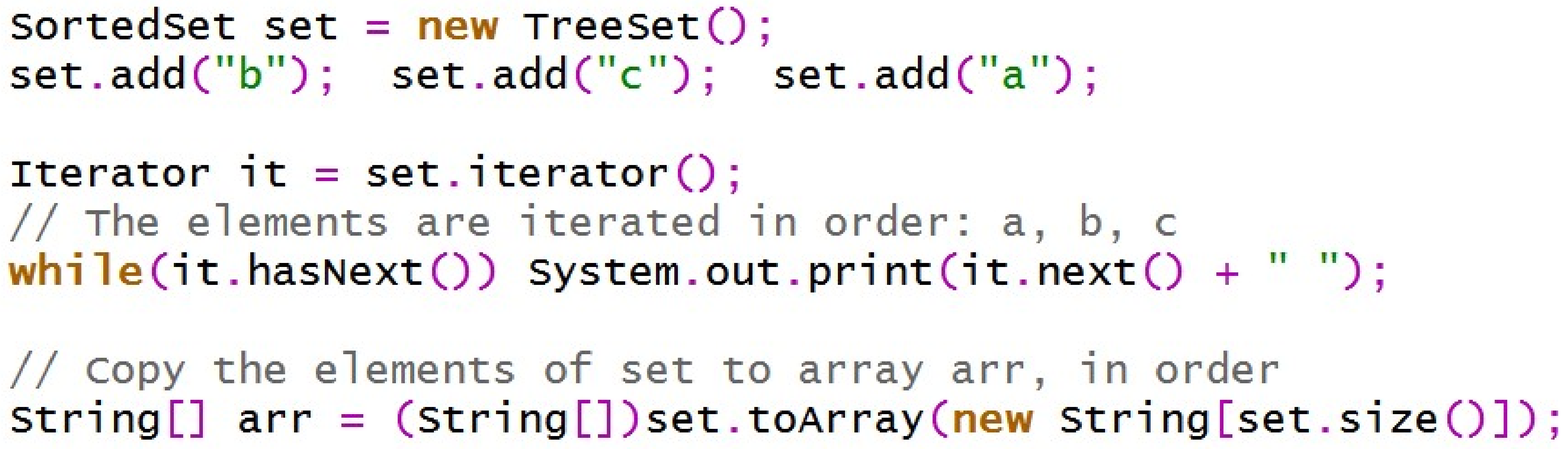
(b) After inserting key 25, (c) after deleting key 4.

71

The Set and SortedSet interfaces

Sets contain no pair of elements e1 and e2 such that e1.equals(e2), and at most one null element.

SortedSet interface extends Set interface.
An implementation of SortedSet is TreeSet.



```
SortedSet set = new TreeSet();
set.add("b");  set.add("c");  set.add("a");

Iterator it = set.iterator();
// The elements are iterated in order: a, b, c
while(it.hasNext()) System.out.print(it.next() + " ");

// Copy the elements of set to array arr, in order
String[] arr = (String[])set.toArray(new String[set.size()]);
```

Note the iterators. YOU CANNOT USE a for(int i=0; i<set.size(); i++) { ... } loop here!

72

Comparing different Sets

```
Set<String> s = new ...;   // HashSet or LinkedHashSet, or TreeSet

s.addAll(Arrays.asList(
  "one two three four five six seven".split(" ")));

if (s instanceof LinkedHashSet)
  System.out.printf("%20s", "LINKED HASH SET: ");
else
  ...
Iterator<String> it = s.iterator();  //  or:
while(it.hasNext())                   // for(String item: s)
  System.out.print(it.next() + " "); //    System.out.print(item + " ");
System.out.println();
```

Output:

```
          HASH SET one two five four three seven six
   LINKED HASH SET one two three four five six seven
          TREE SET five four one seven six three two
```

73

Iterators – objects to navigate over a collection

iterator() method is declared in the interface Collection. It returns an iterator, which is an object of a class implementing Iterator interface.

Other methods from the interface Iterator:

- Object next(); // returns next element in the collection or raises NoSuchElementException
- void remove(); // removes the current element
- boolean hasNext(); // obvious

remove() can be called only once per next():

```
it.next();
it.remove();
it.remove();   // WRONG
                      // IllegalStateException raised
```

74

Associative arrays

A map (e.g., HashMap) keeps a set of pairs (key, value).
Basic methods: put / get.

```
Map<String, String> m = new HashMap<>();
m.put("John Smith", "Ferrari");
m.put("James Bond", "Aston Martin");
m.put("Stefan Karwowski", "Fiat 126P");
System.out.println("James Bond drove " +
         m.get("James Bond"));
```

Standard constructor HashMap() –
default initial capacity (16), default load factor (0.75).
Space / performance tradeoff.

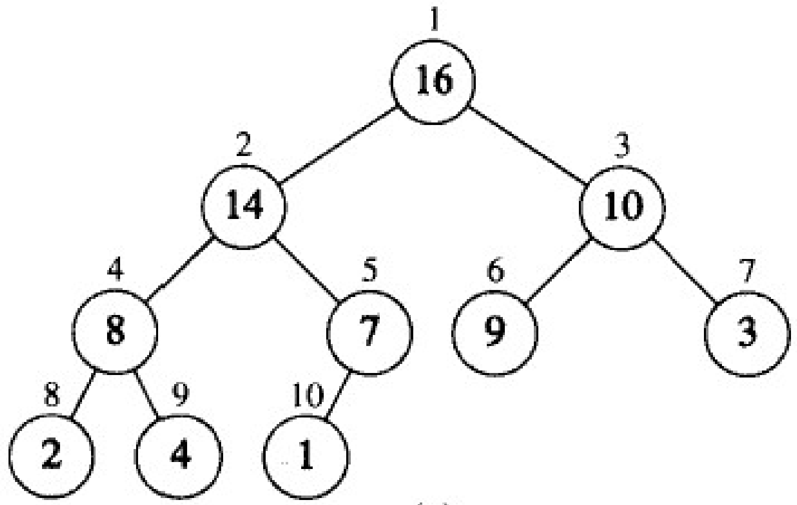
Another constructor: HashMap(int initialCapacity, float loadFactor)

75

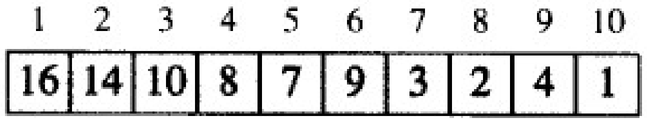
Priority queue

The head of the queue: the least element with respect to the specified ordering; ties are broken arbitrarily.
The queue retrieval operations:
poll, remove, peek, and element (inherited from AbstractQueue) access the element at the head of the queue.
Enqueue / dequeue operations: *O*(log *n*) time.
Search for an element (*contains*(...)) – *O*(*n*) time.

Note what least means here...



(a)



(b)

ConcurrentHashMap

A hash table supporting full concurrency of retrievals
and adjustable expected concurrency for updates.
Multiple threads can read from and write to it
without a possibility of receiving out-of-date or corrupted data.

This class obeys the same functional specification
as Hashtable (and corresponding method names).
However, even though all operations are thread-safe,
retrieval operations do not entail locking,
and the entire table is not getting locked,
but only a portion (by default, of 1/16 size).

ConcurrentHashMap, cont’d

```java
ConcurrentHashMap<String, Integer> myMap = new ConcurrentHashMap<>();
// in Java 7 you can omit <String, Integer> on the right side!
```

What’s the difference between:

```java
if (!myMap.contains("key"))
   myMap.put("key", 3);
```

and

```java
myMap.putIfAbsent("key", 3);
```

?

The first code (with *put*) is unsynchronized: another thread could add a mapping for "key" between the call to method *contains* and the call to *put*.
The second code (with *putIfAbsent*) is correct.

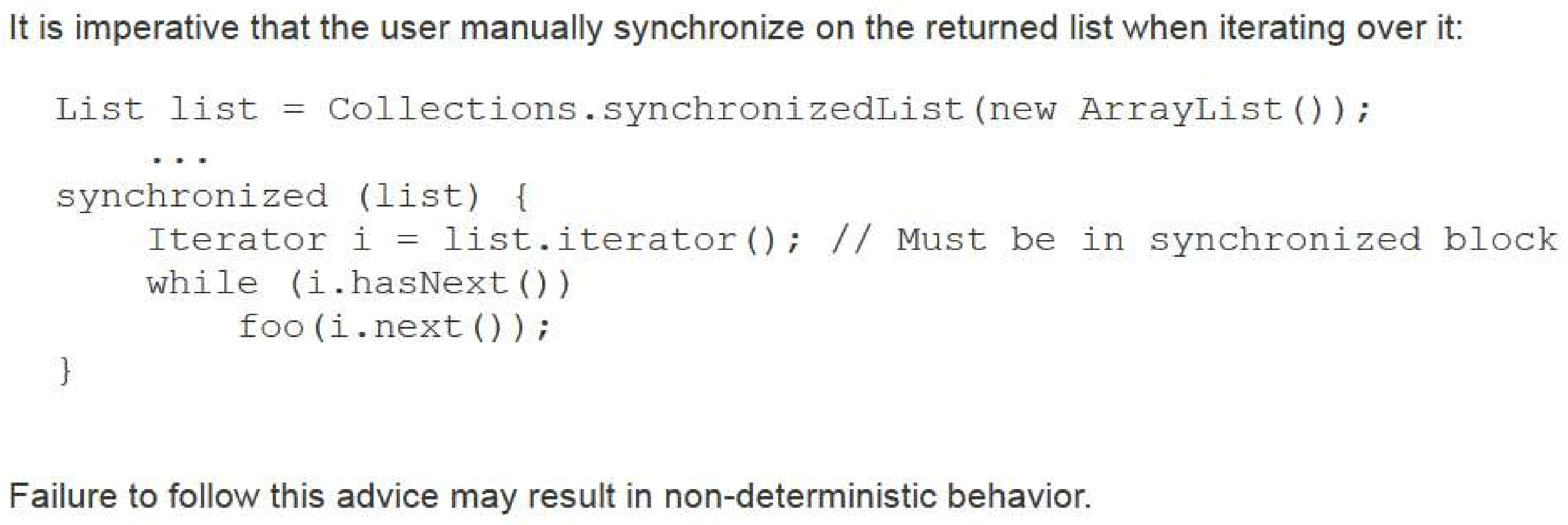
Note: we don’t call the first code thread-unsafe because it does promise to end up in a consistent state afterward.

Collections.synchronized(List|Set|Map)(…)

List<Integer> list = Collections.synchronizedList(new ArrayList<Integer>());

The resulting list (backed by the specified list)
is synchronized (an update from a thread
blocks accesses from all other treads),
but not concurrent.

synchronized = thread-safe


It is imperative that the user manually synchronize on the returned list when iterating over it:

```
List list = Collections.synchronizedList(new ArrayList());
    ...
synchronized (list) {
    Iterator i = list.iterator(); // Must be in synchronized block
    while (i.hasNext())
        foo(i.next());
}
```

Failure to follow this advice may result in non-deterministic behavior.

Create an immutable collection (old Java vs Java 9)

Since Java 5:

```
List<String> values = Arrays.asList("Hello", "World", "from", "Java");
```

Yet no similar factory for a set:

```
Set<String> values = new HashSet<>(Arrays.asList(
"Hello", "World", "from", "Java"));
```

Fortunately in Java 9:

```
List<String> list = List.of("Hello", "World", "from", "Java"); // doesn’t allow null
Set<String> set = Set.of("Hello", "World", "from", "Java");
// Set.of doesn’t allow null or duplicates
```

Similarly for maps in Java 9. For up to 10 entries Maps have overloaded factory methods that take pairs of keys and values:

```
Map<String, Integer> cities = Map.of(
  "Brussels", 1_139000, "Cardiff", 341_000);
```

But no limitation exists with var-args:

```
Map<String, Integer> cities = Map.ofEntries(
   entry("Brussels", 1139000), entry("Cardiff", 341000));
```

New java.util.Arrays methods (Java 9)

```
assertEquals(0, Arrays.mismatch(new int[]{1, 2}, new int[]{2, 2}));
assertEquals(1, Arrays.mismatch(new int[]{1}, new int[]{1, 2}));
assertEquals(1, Arrays.mismatch(new int[]{1, 3}, new int[]{1, 2}));
assertEquals(-1, Arrays.mismatch(new String[]{"a", "ab"},
          new String[]{"a", "ab"}));

assertEquals(-1, Arrays.mismatch(
     new int[]{1, 3}, 0, 1,
     new int[]{1, 2}, 0, 1)
);  // extra (start, end) parameters!
```

Also new: Arrays.compare
and Arrays.equals over a specified range

Randomized iteration order (Java 9)

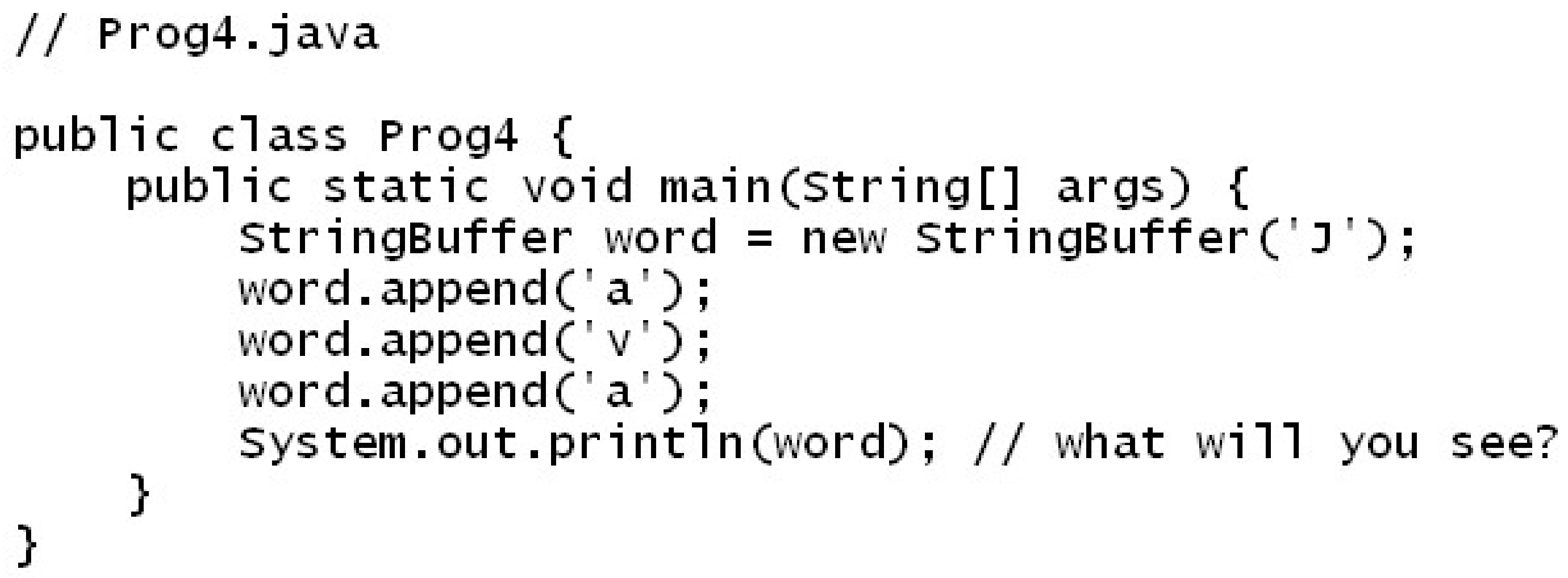
[http://iteratrlearning.com/java9/2016/11/09/java9-collection-factory-methods.html]

HashSet and HashMap iteration order has always been unspecified, but fairly stable, leading to code having inadvertent dependencies on that order. This causes things to break when the iteration order changes, which occasionally happens.

The new Set/Map collections (from Java 9) change their iteration order from run to run, hopefully flushing out order dependencies earlier in test or development.

A little puzzle

[http://www.csc.uvic.ca/~csc330/]



```
// Prog4.java

public class Prog4 {
    public static void main(String[] args) {
        StringBuffer word = new StringBuffer('J');
        word.append('a');
        word.append('v');
        word.append('a');
        System.out.println(word); // what will you see?
    }
}
```

"Java"? No...

Why not?

83

Puzzle, solution

[http://www.csc.uvic.ca/~csc330/Assignment1/Ass1-answers.pdf]

There are four constructors for the class StringBuffer.
None of the constructors accepts a parameter of type char,
which is what the program uses.
However, in Java a value of type char can be coerced to
a value of type int.

So the instantiation is equivalent to new StringBuffer(74)
because 'J' has the ASCII code 74.
That particular invocation creates a new StringBuffer
which is empty but which has an initial capacity of 74 characters.

☺

(But later, ‘a’ etc. will be appended, so the output is: ava
(beware, word.length() == 3, but word.capacity() == 74).

84

Whence exceptions?

Throw exceptions only in truly exceptional situations.
Not e.g. when not finding an element in an array
(this is normal and you can return –1 as a (pseudo)index).

Consider a stack data structure. The contract of the *pop* method requires that the stack not be empty when it is called, so if you try to pop something from an empty stack, that’s exceptional.
There is no meaningful value to return in this case, and you shouldn’t have made the call in the first place.
Similar logic applies to the top and push methods.

85

FixedStack class example

```
class FixedStack {
    private int capacity;
    private int size;
    private Object[] data;
    ...
    public Object pop() throws StackException {
        if (size <= 0)  throw new StackException("underflow");
        return data[--size];
    }
    public void push(Object o) throws StackException  {
        if (size == capacity) throw new StackException("overflow");
        data[size++] = o;
    }
    ...
}
```

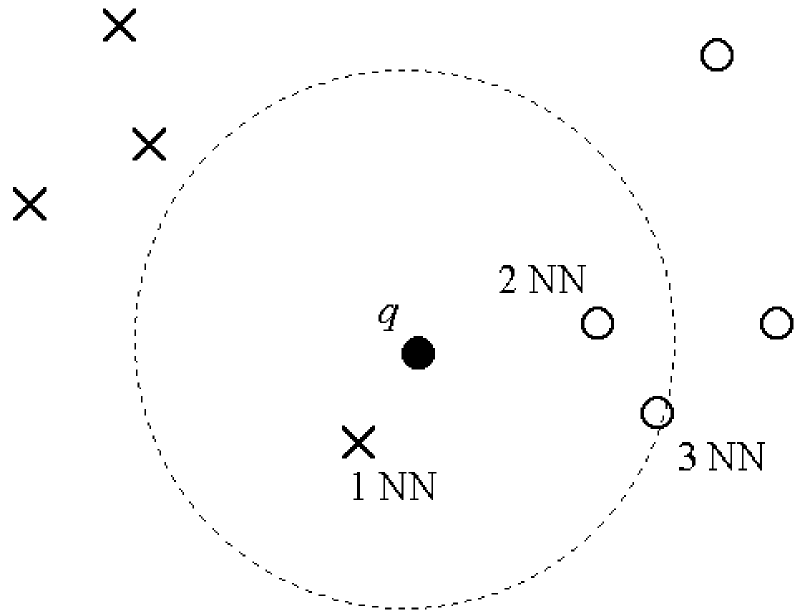
```
class StackException extends Exception {
    StackException() {}
    StackException(String msg) { super(msg); }
}
```

86

Heap applications, cont’d

k nearest neighbors search

(basic task in pattern recognition / basic classification method)



k-NN rule for *k* = 3. Query *q* assigned to the class of “circles”

Finding the *k* nearest neighbors

Naïve method: $O(nk)$ worst case time
(albeit close to $O(n)$ in practice).

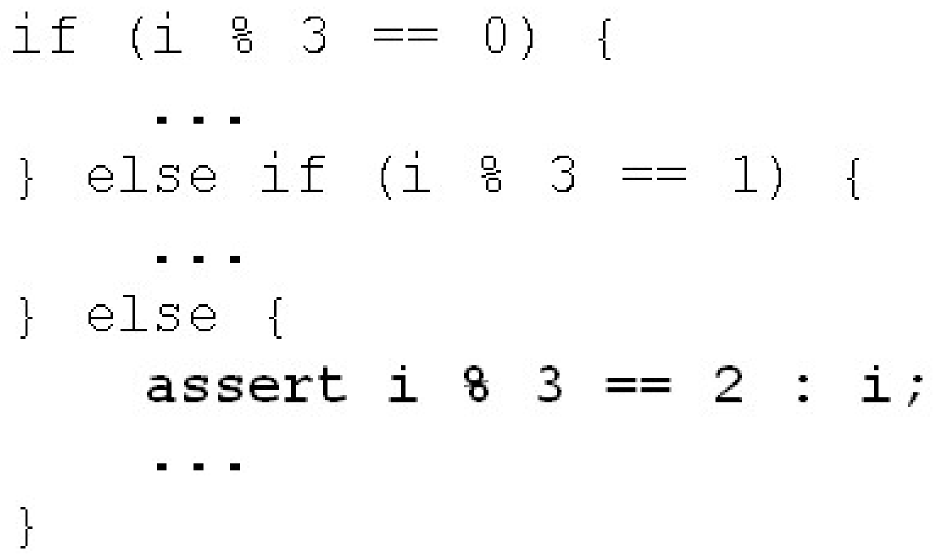
Idea: use a sorted list of *k* (currently)
nearest neighbors of *q*, update it if needed in $O(k)$ time.
Worst case: each inspected sample is among
the *k* nearest (so far) neighbors of *q*.

With a heap: $O(n \log k)$ worst case time.
But may be slower than naïve in practice...
unless *k* really large.

Idea: store the *k*-th neighbor (i.e., the farthest of those
k nearest neighbors) in the root. Testing a new candidate: one
comparison and then (at worst) heapify() in $O(\log k)$ time.

Funny example

[http://download.oracle.com/javase/1.4.2/docs/guide/lang/assert.html]



```
if (i % 3 == 0) {
    ...
} else if (i % 3 == 1) {
    ...
} else {
    assert i % 3 == 2 : i;
    ...
}
```

Actually the assertion will fail many times,
if *i* can be negative...
(% in Java: remainder, may be negative).

Cleanup, finalize()

Two ways: finalization and garbage collection.

finalize() in the Object class:
When the garbage collector is ready to release the storage for your object, it will first call finalize(), and only on the next garbage-collection pass will it reclaim the object's memory.

In C++ we have a destructor and objects always get destroyed (immediately); not so in Java.
(Perhaps an object will be destroyed, i.e. its storage will be garbage-collected (when? We don't know), or perhaps not... – to save time.)

1. Your objects might not get garbage collected.
2. Garbage collection is not destruction.

So, what do we need finalize() for??

We may allocate space using *native methods*,
i.e. calling non-Java code from Java.
Example: allocate memory with C’s malloc().
Until free() called, storage will not be released.

Of course, free() is a C/C++ function, so
we need to call it in a native method inside our finalize().

But I won’t use that ugly C or C++ in my Java code!
So, I will never need finalize(), right?

Not quite true...

91

finalize(), interesting use

[B.Eckel, *Thinking in Java*]

Verification of the termination condition of an object.

When we're no longer interested in an object,
it should be in a state whereby its memory can be
safely released.
E.g. the object is an open file, that file should be
explicitly closed before the object is garbage collected.

The value of finalize() is that it can be used to
eventually discover this condition, even if it isn't always
called. If one of the finalizations happens to reveal the
bug, then you discover the problem.

92

finalize(), interesting use, cont'd Code example (1/2) [B.Eckel, *Thinking in Java*]

```java
//: c04:TerminationCondition.java
// Using finalize() to detect an object that
// hasn't been properly cleaned up.
import com.bruceeckel.simpletest.*;

class Book {
  boolean checkedOut = false;
  Book(boolean checkOut) {
    checkedOut = checkOut;
  }
  void checkIn() {
    checkedOut = false;
  }
  public void finalize() {
    if(checkedOut)
      System.out.println("Error: checked out");
  }
}
```

93

finalize(), interesting use, cont'd
Code example (2/2) [B.Eckel, *Thinking in Java*]

```
public class TerminationCondition {
  static Test monitor = new Test();
  public static void main(String[] args) {
    Book novel = new Book(true);
    // Proper cleanup:
    novel.checkIn();
    // Drop the reference, forget to clean up:
    new Book(true);
    // Force garbage collection & finalization:
    System.gc();
    monitor.expect(new String[] {
      "Error: checked out"}, Test.WAIT);
  }
} ///:~
```

94

Adapters, example

```
interface R2D2 {
  String display();
  String show();
  int calculate(int a, int b);
  float generate();
  double mean();
}
```

```
abstract class R2D2Adapter implements R2D2 {
  public String display() { return ""; }
  public String show() { return ""; }
  public int calculate(int a, int b) { return 0; }
  public float generate() { return 0.0f; }
  public double mean() { return 0.0d; }
}
```

```
public class Robot extends R2D2Adapter {
  public String display() {
    String s = "Adapting R2D2 to Robot";
    System.out.println(s);
    return s;
  }
  public static void main(String args[]) {
    Robot r = new Robot();     r.display();
  }
}
```

95

null stuff

[based on http://norvig.com/java-iaq.html]

Is null an object?

No. Because (null instanceof Object) is false.

Can I call a method on null, like

```
x = null;
x.m();
```

?

No (if m is a non-static method).

There is a single null, not one per class,
i.e. for example ((String)null == (Date)null).

Assertions, more formally

[http://download.oracle.com/javase/1.4.2/docs/guide/lang/assert.html]

More formally:
assert *expr1*;
or: assert *expr1 : expr2;*

expr1 is a condition (boolean) to check.
If it is false, we have a failed assertion
→ AssertionError exception.

expr2 is an optional expression having a value
(invocation of a void method is forbidden).
The system passes the value of expr2 to
an appropriate AssertionError constructor,
which uses the string representation of the value
as the error's detail message.

Applications of priority queues

- Maintaining *k*-best list (finding *k* nearest neighbors – next slide).
- Graph algorithms (Prim's alg for the minimum spanning tree problem; Dijkstra's alg for the single-source shortest-path problem, etc.).
- Event simulations.
- Sorting (heap sort and variants of).
- Multi-way merging (e.g., many large sorted files).
- Huffman coding.

java.lang.Error, java.lang.RuntimeException

[members.tripod.com/~psdin/comlang/basicJava.PDF]

java.lang.Error:

- LinkageError (→ NoClassDefFoundError …), VirtualMachineError (→ OutOfMemoryError, StackOverflowError ...), AWTError, ...;
- usually irrecoverable;
- don't try to catch them.

java.lang.RuntimeException:

- ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException, ClassCastException, ...
- often due to programmatic errors, rather than special runtime conditions.

99

Try-finally for file closing: old style

```
import java.io.*;
class ProcessFile {
   public static void main(String[] args) {
      if (args.length > 0) {
         FileReader f = null;
         try { // Next line: open a file
            f = new FileReader(args[0]);
            SomeOtherClass.process(f);
            ~~f.close();~~
         }
         catch (IOException x) {
            System.out.println(x);
            ~~if (f != null)~~
               ~~f.close();~~
         }
      }
   }
}
```

Problems?

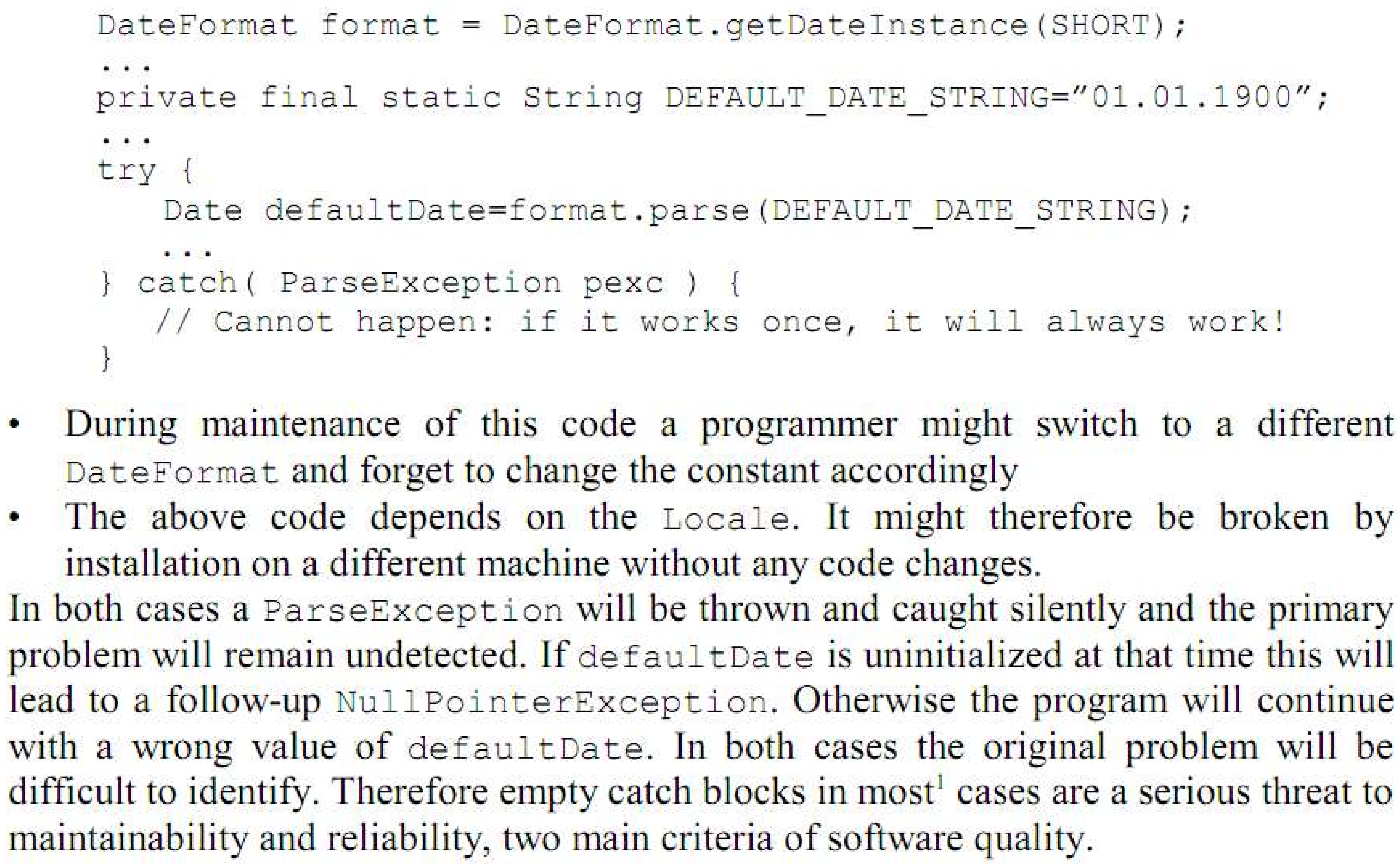
f.close() occurs twice in the code – unelegant

Nicer.

```
finally {
   if (f != null)
      f.close();
}
```

100

Empty catch blocks?



```
DateFormat format = DateFormat.getDateInstance(SHORT);
...
private final static String DEFAULT_DATE_STRING="01.01.1900";
...
try {
   Date defaultDate=format.parse(DEFAULT_DATE_STRING);
   ...
} catch( ParseException pexc ) {
   // Cannot happen: if it works once, it will always work!
}
```

- During maintenance of this code a programmer might switch to a different `DateFormat` and forget to change the constant accordingly
- The above code depends on the `Locale`. It might therefore be broken by installation on a different machine without any code changes.

In both cases a `ParseException` will be thrown and caught silently and the primary problem will remain undetected. If `defaultDate` is uninitialized at that time this will lead to a follow-up `NullPointerException`. Otherwise the program will continue with a wrong value of `defaultDate`. In both cases the original problem will be difficult to identify. Therefore empty catch blocks in most[1] cases are a serious threat to maintainability and reliability, two main criteria of software quality.

http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.11.4355&rep=rep1&type=pdf

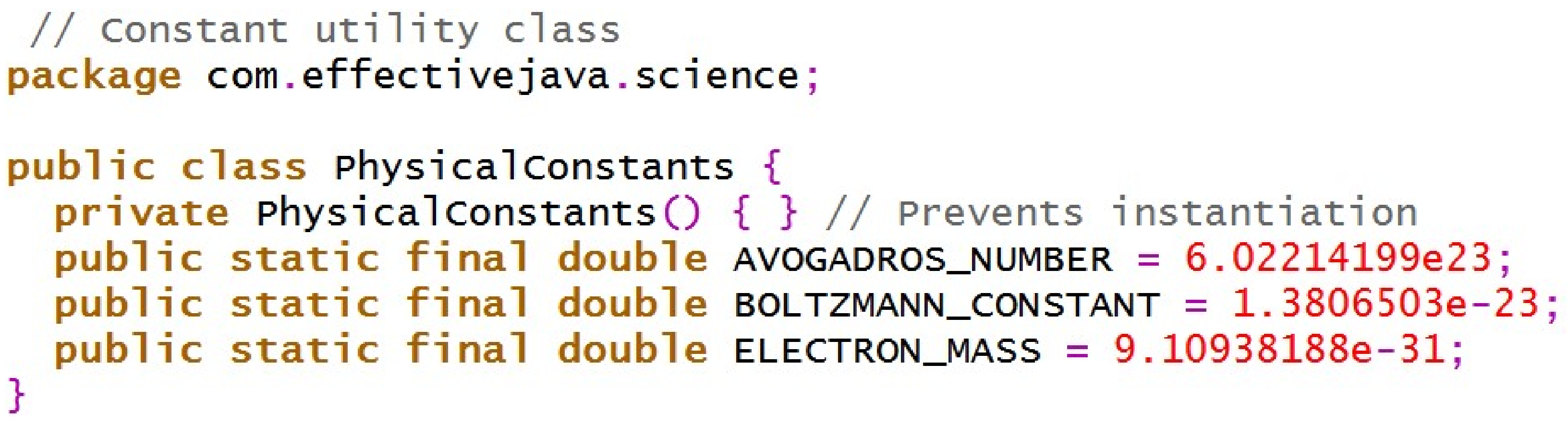
Use interfaces only to define types [*Effective Java*, Item 19]

That a class uses some constants internally is an implementation detail. Implementing a constant interface causes this implementation detail to leak into the class's exported API.

If in a future version the class is modified and no longer needs to use the constants, it still must impl. the interface to ensure binary compatibility.

If a non-final class implements a constant interface, all of its subclasses will have their namespaces polluted by the constants in the interface.

Recommended use (alternatively: an enum):



```java
 // Constant utility class
package com.effectivejava.science;

public class PhysicalConstants {
  private PhysicalConstants() { } // Prevents instantiation
  public static final double AVOGADROS_NUMBER = 6.02214199e23;
  public static final double BOLTZMANN_CONSTANT = 1.3806503e-23;
  public static final double ELECTRON_MASS = 9.10938188e-31;
}
```

Associative arrays, internally

Standard constructor HashMap() –
default initial capacity (16) and
default load factor (0.75).

Capacity: the # of buckets in the hash table.

Load factor: how full the hash table is allowed
to be before its capacity is automatically increased.

Load factor too large: insert / access / delete
may be too costly on avg.

Other constructors exist, e.g.
HashMap(int initialCapacity, float loadFactor)