Zaawansowane programowanie obiektowe

Lecture 4

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

Łódź, 2018

Annotations

Annotations are similar to meta tags;
they prepend
package declarations, type (class) declarations,
constructors, methods, fields...
Smth like comments not only for humans,
but also for the compiler (and to some degree
also for the JVM) and for processing tools.

The Java compiler generates the same VM instructions
with or without the annotations.

@Override, @Deprecated

@Override: to denote that the annotated method is to override a method from a superclass.
You forgot *public* in front of it? Compile error!

```
public class Overrode {
   @Override
   public int hashcode() {
      return 0;
   }

   @Override
   public boolean equals(Object o) {
      return true;
   }
}
```

```
> javac Overrode.java

Overrode.java:2: method does not override a method from its
superclass
      @Override
       ^
1 error
```

http://www.java-tips.org/java-se-tips/java.lang/introducing-annotations.html

```
public class Main {

   /**
    * @deprecated Out of date. Use System.foobar() instead.
   */

   @Deprecated
   public static void deprecatedMethod() {
      System.out.println("Don't call me");
   }
}
```

Annotation elements

[based on C.Horstmann, *Core Java for the Impatient*, Chap. 11]

Annotations can have key/value pairs
called elements.
null values are forbidden.
Values are: primitive type values, strings,
enum instances, Class objects, and even annotations
(and arrays of the preceding, but not arrays of arrays).

```
@BugReport(showStopper = true,
    assignedTo = "Harry",
    testCase = CacheTest.class,
    status = BugReport.Status.CONFIRMED)
```

Annotation elements, cont'd

[based on C.Horstmann, *Core Java for the Impatient*, Chap. 11]

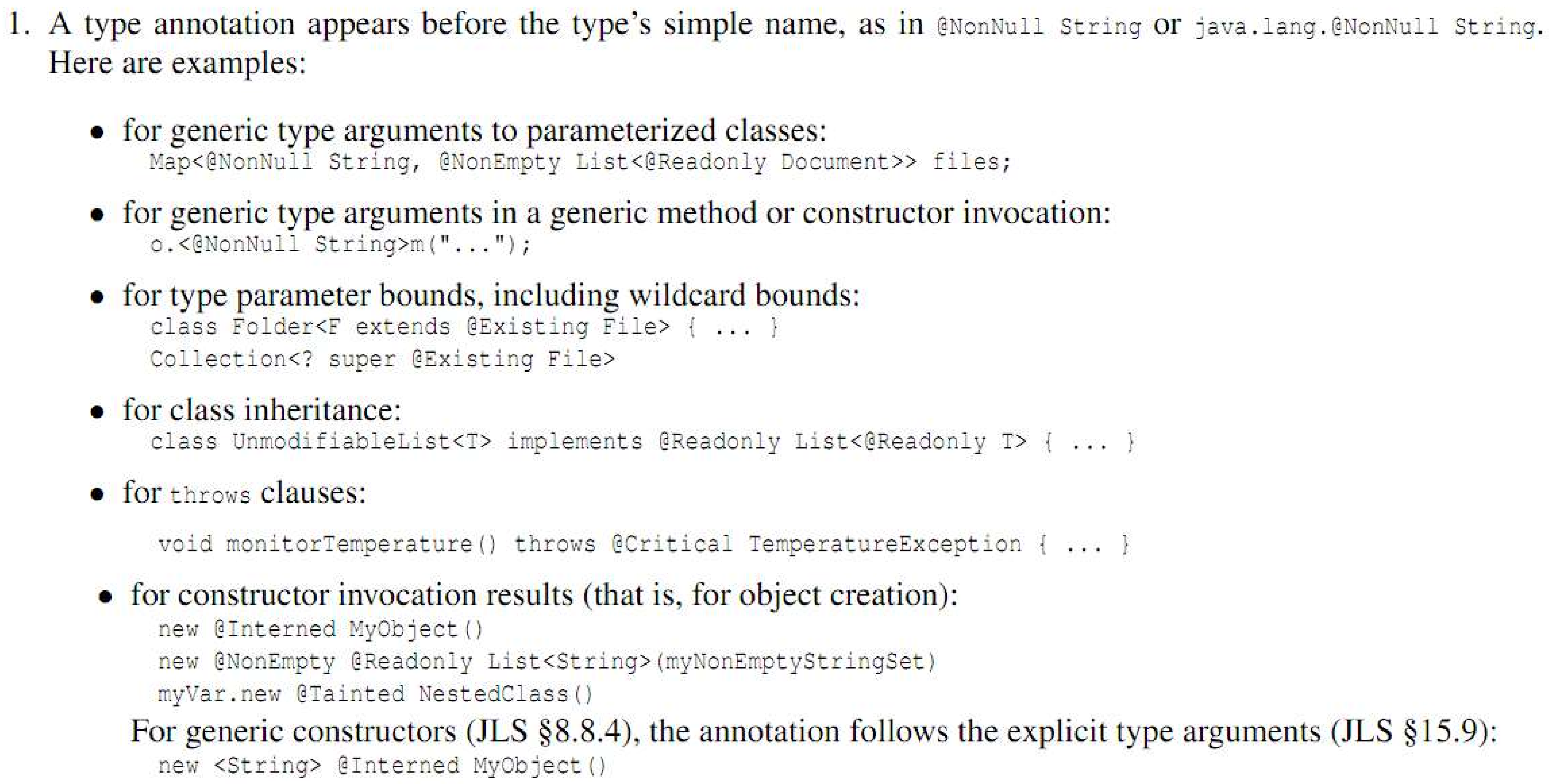
Elements can have default values.
E.g. the timeout element of the JUnit @Test annotation has default 0L.
I.e. @Test is equivalent to @Test(timeout=0L).

Shortcuts:

1. If the element name is value, and that is the only element specified, you can omit value=.
E.g., @SuppressWarnings("unchecked") is the same as @SuppressWarnings(value="unchecked").

2. Arrays are passed in braces, e.g.
@BugReport(reportedBy = {"Harry", "Fred"}).
Can be shortened if the array has a single component:
@BugReport(reportedBy = "Harry") // Same as {"Harry"}

Type annotations in Java 8



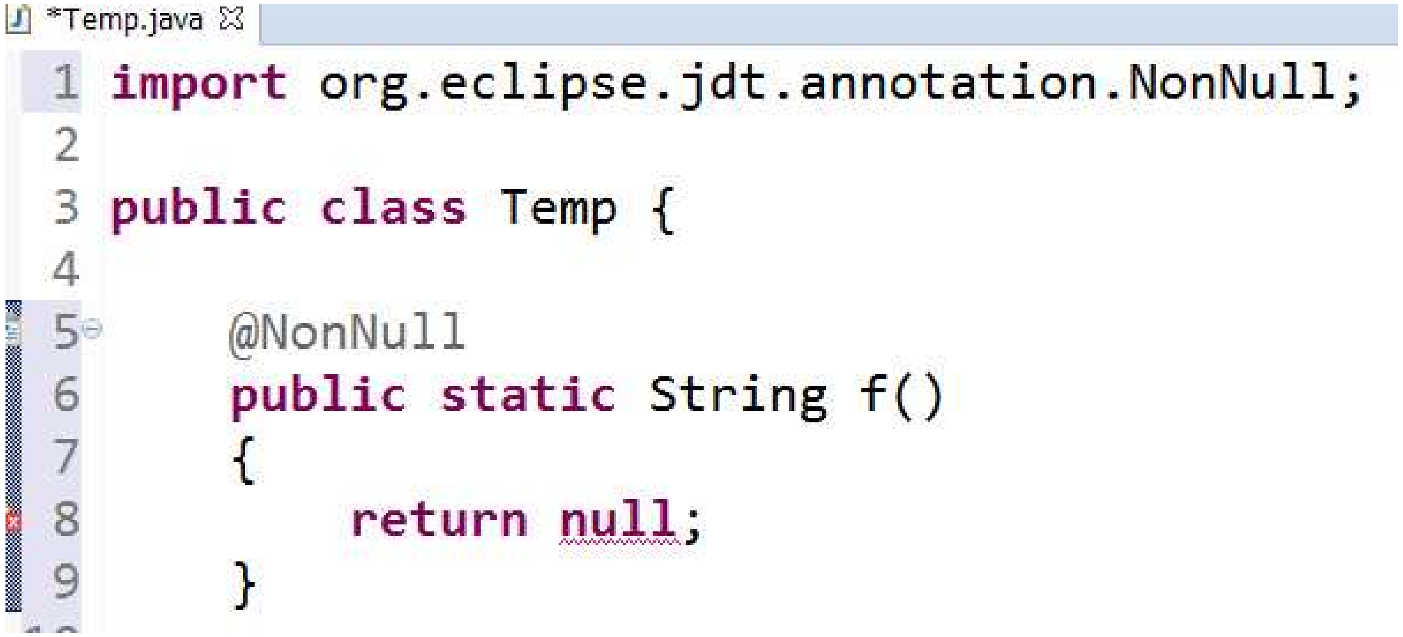
1. A type annotation appears before the type's simple name, as in `@NonNull String` or `java.lang.@NonNull String`. Here are examples:

   - for generic type arguments to parameterized classes:
     ```
     Map<@NonNull String, @NonEmpty List<@Readonly Document>> files;
     ```
   - for generic type arguments in a generic method or constructor invocation:
     ```
     o.<@NonNull String>m("...");
     ```
   - for type parameter bounds, including wildcard bounds:
     ```
     class Folder<F extends @Existing File> { ... }
     Collection<? super @Existing File>
     ```
   - for class inheritance:
     ```
     class UnmodifiableList<T> implements @Readonly List<@Readonly T> { ... }
     ```
   - for `throws` clauses:
     ```
     void monitorTemperature() throws @Critical TemperatureException { ... }
     ```
   - for constructor invocation results (that is, for object creation):
     ```
     new @Interned MyObject()
     new @NonEmpty @Readonly List<String>(myNonEmptyStringSet)
     myVar.new @Tainted NestedClass()
     ```
     For generic constructors (JLS §8.8.4), the annotation follows the explicit type arguments (JLS §15.9):
     ```
     new <String> @Interned MyObject()
     ```

… and more.

http://download.oracle.com/otn-pub/jcp/annotations-2013_03_15-edr3-spec/
2013-03-15-jsr308-spec.pdf?AuthParam=1374602737_cbcc705c285f1bf0dc349ee9b8e1db0b

@NonNull (type annotation, Java 8)


*Temp.java

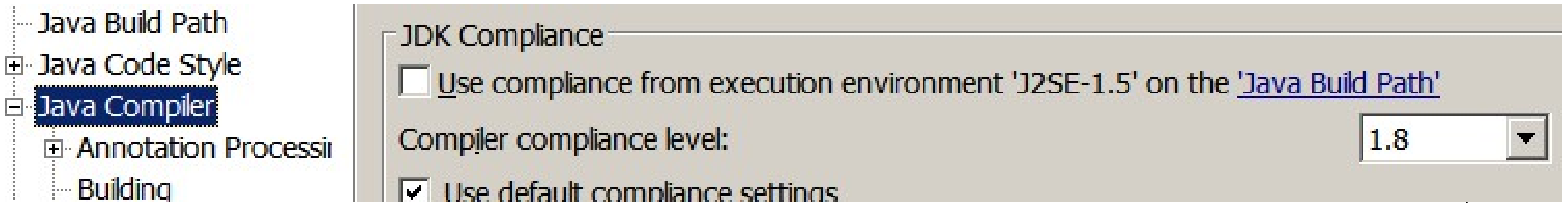
```
import org.eclipse.jdt.annotation.NonNull;

public class Temp {

    @NonNull
    public static String f()
    {
        return null;
    }
```

Null type mismatch: required '@NonNull String' but the provided value is null

Eclipse 4.6 (Neon): Properties / Java Compiler: set compliance level to 1.8:



JDK Compliance

☐ Use compliance from execution environment 'J2SE-1.5' on the 'Java Build Path'

Compiler compliance level: 1.8

☑ Use default compliance settings

Generics

When you take an element out of a Collection, you must cast it to the type of element that is stored in the collection → inconvenient and unsafe.

The compiler does not check that your cast is the same as the collection’s type, so the cast can fail at run time.

With generics, the compiler will know the object type, so that it can be checked.

```
ArrayList<Integer> listOfIntegers; // new syntax: <TYPE_NAME>
Integer integerObject;
listOfIntegers = new ArrayList<Integer>();
listOfIntegers.add(new Integer(10)); // Can only pass in Integer objects
integerObject = listOfIntegers.get(0); // no cast required
```

8

Defining generics

[http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf]

Small excerpt from the definitions of the interfaces List and Iterator in package java.util:

```
public interface List<E> {
  void add(E x);
  Iterator<E> iterator();
}

public interface Iterator<E> {
  E next();
  boolean hasNext();
}
```

Might seem very similar to C++ templates so far...

...But it is different than C++ templates

In C++, templates are kind of intelligent macros.
Having vector<int> and vector<double> in our program, we also have two copies of the code: a single copy for each used type (=template parameter).
In other words, it is a compile-time mechanism.

Not so in Java. There aren't multiple copies of the code: not in source or in memory etc. An analogy to plain methods: parameters in methods are *values* of a given type; parameters of generics are *types*.

A method has *formal value parameters* that describe the kinds of values it operates on; a generic declaration has formal type parameters.
When a method is invoked, *actual arguments* are substituted for the formal parameters, and the method body is evaluated.
When a generic declaration is invoked, the actual type arguments are substituted for the formal type parameters.

10

Consequently...

...the following piece of code:

```
List <String> l1 = new ArrayList<String>();
List<Integer> l2 = new ArrayList<Integer>();
System.out.println(l1.getClass() == l2.getClass());
```

...prints true.

This is so-called type erasure.
When you define a generic type, a corresponding raw type is automatically provided;
e.g. ArrayList<String> → ArrayList,
Pair<Integer> → Pair,
Pair<String, Integer> → Pair.
When you then call a generic method, the compiler then inserts casts where necessary.

11

Restrictions on generics

[Core Java, vol. 1, Chap. 12]

Type params cannot be instantiated with primitive types.
Bad: Pair<int>, Good: Pair<Integer>

Cannot throw or catch instances of a generic class.
In particular, a generic class can't extend Throwable.

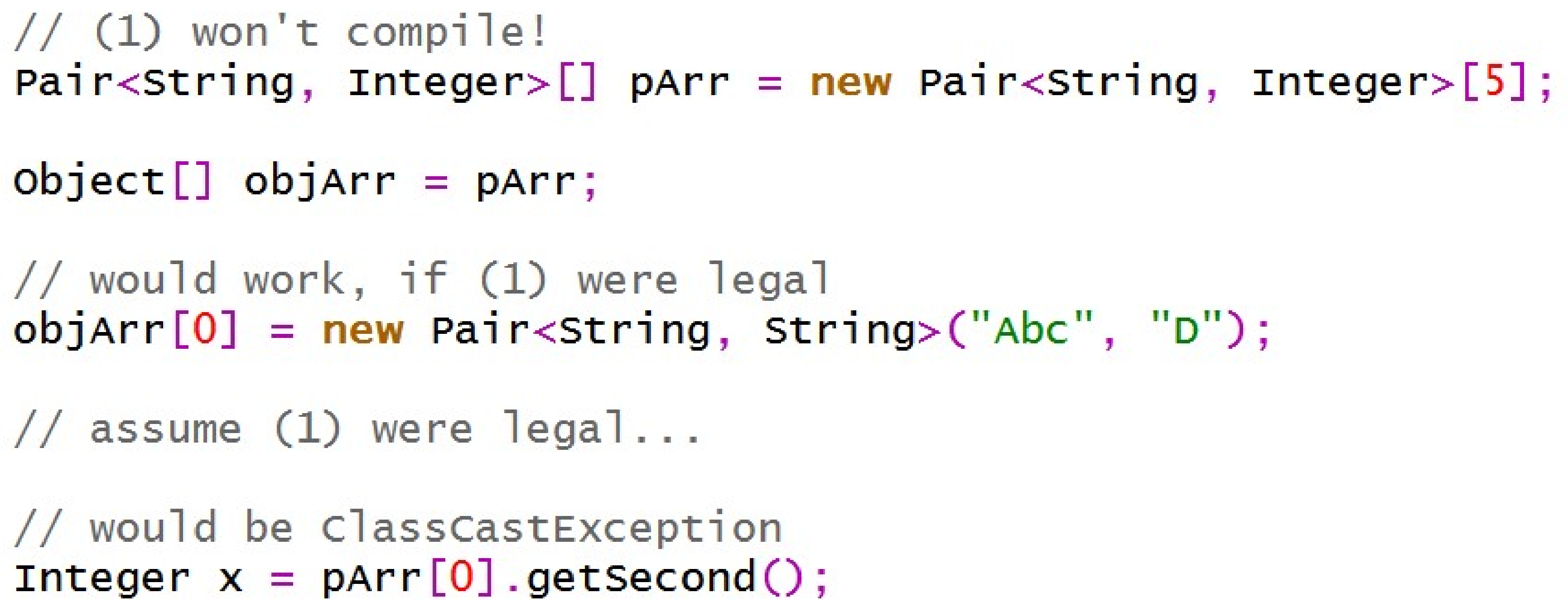
Arrays of parametrized types are not legal.
Bad: Pair<String>[ ] table = new Pair<String>[10];

Cannot instantiate type variables.
Bad: new T(...), new T[...], T.class

(and more, see [Core Java, vol. 1])

Why arrays of (concrete) parametrized types are illegal

[http://edu.pjwstk.edu.pl/wyklady/zap/scb/W2/W2.htm]



```
// (1) won't compile!
Pair<String, Integer>[] pArr = new Pair<String, Integer>[5];

Object[] objArr = pArr;

// would work, if (1) were legal
objArr[0] = new Pair<String, String>("Abc", "D");

// assume (1) were legal...

// would be ClassCastException
Integer x = pArr[0].getSecond();
```

Alternatives:

- arrays of raw types (unsafe),
- arrays of T<?>,
- collections (best choice).

Creating a generic method inside an ordinary class

[based on Core Java, vol. 1, Chap. 12]

```
class ArrayAlg
{
   public static <T> T getMiddle(T[] a)
   {
      return a[a.length / 2];
   }
}

...

String[] names = {"John", "Q.", "Public"};
String middle = ArrayAlg.<String>getMiddle(names);
```

The invocation of getMiddle(names) may look strange
but is safe. In almost all cases (also here), a shorter form works too:
String middle = ArrayAlg.getMiddle(names);

Subtyping with generics

[M. Naftalin & Ph. Wadler, *Java Generics and Collections*, 2006, Chap. 2]

`Integer`	is a subtype of	`Number`
`Double`	is a subtype of	`Number`
`ArrayList<E>`	is a subtype of	`List<E>`
`List<E>`	is a subtype of	`Collection<E>`
`Collection<E>`	is a subtype of	`Iterable<E>`

```java
List<Number> nums = new ArrayList<Number>();
nums.add(new Integer(2));  // or: nums.add(2);
nums.add(new Double(3.14)); // or: nums.add(3.14);
assert nums.toString().equals("[2, 3.14]") : "Error!";
```

Replace the code above with this:

```java
List<Integer> ints = Arrays.asList(1, 2);
List<Number> nums = ints; // compile-time error!
nums.add(3.14);
assert ints.toString().equals("[1, 2, 3.14]"); // ?!?!
```

Generic collections detect more errors than arrays at compile-time

[http://www.inf.ed.ac.uk/teaching/courses/tpl/lectures/java-generics.pdf]

`ArrayStoreExceptions` become compile errors

- Arrays:

```
Integer[] ints = new Integer[]{1,2,3}
Number[] nums = ints;
nums[2] = 3.14;                  // run-time error
```

`Integer[]` **is** a subtype of `Number[]`

- Collections:

```
List<Integer> ints = Arrays.asList(1,2,3);
List<Number> nums = ints;  // compile-time error
nums.put(2, 3.14);
```

`List<Integer>` **is not** a subtype of `List<Number>`

Prefer (generic) lists over arrays
(J. Bloch, *Effective Java, 2nd ed.*, Item 25)

```
// Fails at runtime!
Object[] objectArray = new Long[1];
objectArray[0] = "bad data"; // throws ArrayStoreException
```

```
// Won't compile!
List<Object> ol =
  new ArrayList<Long>(); // ERROR! Incompatible types
ol.add("bad data");
```

Conclusion: better to have a compile error,
so prefer a list.

List<Object> list = … is better than List list = …

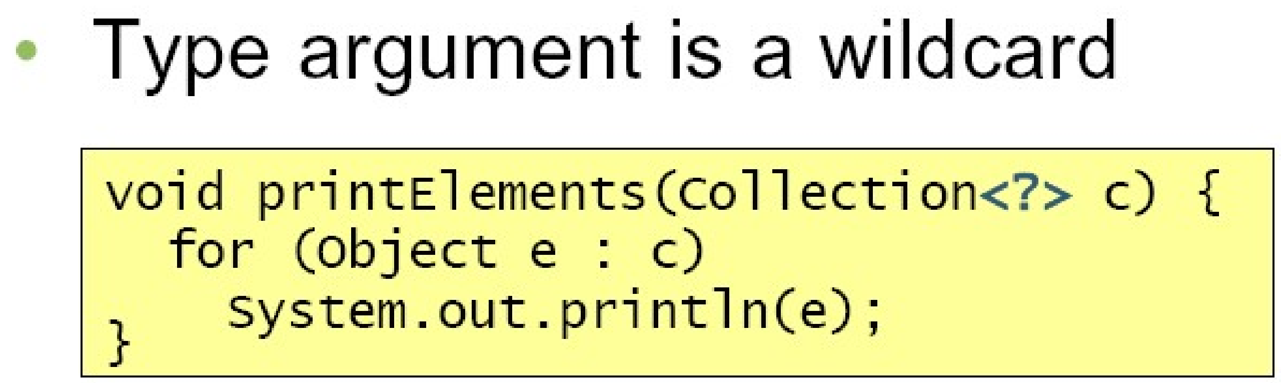
It is fine to use types that are parameterized to allow insertion of arbitrary objects, such as List<Object>. But why List<Object> is better than List?

Because (as you know already) List<String> is
a subtype of the raw type List, but not of List<Object>.

As a consequence, you lose type safety
if you use a raw type like List,
but not if you use a parameterized type
like List<Object>.

Based on [J. Bloch, *Effective Java, 2nd ed.*], Item 23.

Wildcards



- Type argument is a wildcard

```
void printElements(Collection<?> c) {
  for (Object e : c)
    System.out.println(e);
}
```

A wildcard denotes a representative from a family of types.

Unbounded wildcard ?
(all types)

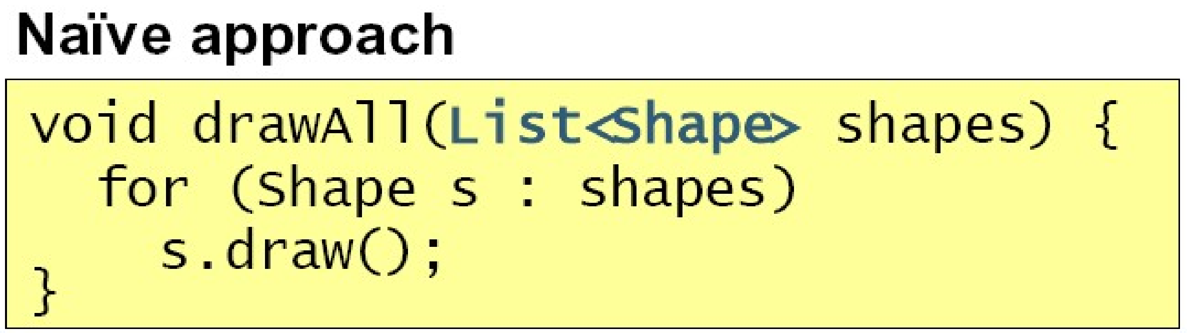
Upper-bounded wildcard ? extends supertype
(all types that are subtypes of supertype)

Lower-bounded wildcard ? super subtype
(all types that are supertypes of subtype)

Bounded wildcard example

[http://www.angelikalanger.com/Conferences/Slides/JavaGenerics-JavaOne-2005.pdf]

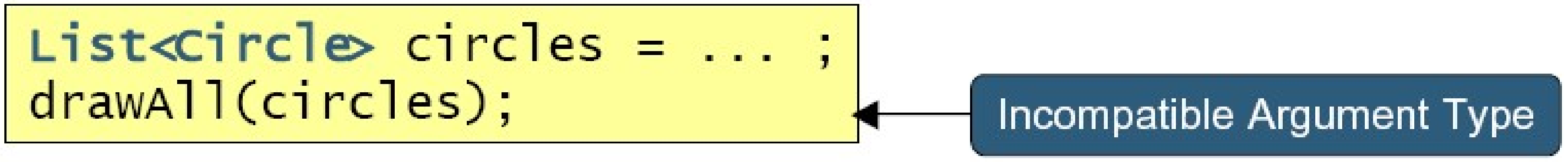
Consider a method that draws objects from a class hierarchy of shapes.



Naïve approach

```
void drawAll(List<Shape> shapes) {
  for (Shape s : shapes)
    s.draw();
}
```

Cannot draw e.g. a list of circles
because List<Circle> is NOT a subtype of List<Shape>.



```
List<Circle> circles = ... ;
drawAll(circles);
```

Bounded wildcard example, cont’d

Fix
with ‘?’ ??

```
void drawAll(List<?> shapes) {
  for (Shape s : shapes)
    s.draw();
}
```

Error: ? Does Not Have a Draw Method

- Compiler needs more information about “unknown” type

Use upper-bound wildcard to solve the problem.

- "? Extends shape" stands for “any subtype of shape”
  - Shape is the *upper bound* of the bounded wildcard
- Collection<? Extends shape> stands for “collection of any kind of shapes”
  - Is the supertype of *all* collections that contain shapes (or subtypes thereof)

```
void drawAll(List<? extends Shape> shapes) {
  for (Shape s : shapes)
    s.draw();
}
```

```
List<Circle> circles = ... ;
drawAll(circles);
```

fine

addAll(...) in Collection<E>

[M. Naftalin & Ph. Wadler, *Java Generics and Collections*, 2006, Chap. 2]

```
interface Collection<E> {
  ...
  public boolean addAll(Collection<? extends E> c);
  ...
}
```

```
List<Number> nums = new ArrayList<Number>();
List<Integer> ints = Arrays.asList(1, 2);
List<Double> dbls = Arrays.asList(2.78, 3.14);
nums.addAll(ints);
nums.addAll(dbls);
assert nums.toString().equals("[1, 2, 2.78, 3.14]");
```

nums.addAll(ints) is OK since nums has type List<Number>,
which is a subset of Collection<Number>
and ints has type List<Integer>,
which is a subtype of Collection<? extends Number>.
(Similarly with dbls.)

Using wildcards when declaring variables

[M. Naftalin & Ph. Wadler, *Java Generics and Collections*, 2006, Chap. 2]

```
List<? extends Number> nums1 = Arrays.asList(5.3, 6);
System.out.println(nums1.get(0).getClass()); // class java.lang.Double
System.out.println(nums1.get(1).getClass()); // class java.lang.Integer

List<? extends Number> nums2 = Arrays.asList(5.3, 6.0);
System.out.println(nums2.get(0).getClass()); // class java.lang.Double
System.out.println(nums2.get(1).getClass()); // class java.lang.Double

nums2.add(3.14);  // compile-time error
```

We can't add a double to a list of "? extends Number"
since it might be a list of some other subtype of Number.

(To add items we need another wildcard, ? super ...)

Wildcards with supertype bounds (*? super ...*):
write to a generic object;

with subtype bounds (*? extends ...*):
read from a generic object.

Intro to I/O

The (main) hierarchy of input / output classes starts with abstract classes InputStream and OutputStream.

Basic classes for reading and writing sequences of bytes are FileInputStream and FileOutputStream.
When we create a file object (eg. of any of the two just mentioned classes), the file is open.

```
FileInputStream inp = new FileInputStream("1.dat");
FileOutputStream outp = new FileOutputStream("2.dat");
```

Read a byte (0..255), write a byte:
`int read();  write(int);`
Read value –1 signals the end of file.

24

Byte-by-byte copy example ([Barteczko'04])

```
import java.io.*;

class CopyFile
{
  static public void main(String[] args)
  {
    FileInputStream inp;
    FileOutputStream outp;
    try {
      inp = new FileInputStream(args[0]);
      outp = new FileOutputStream(args[1]);
      int c;
      while ((c=inp.read()) != -1) outp.write(c);  // actual copying
    } catch(ArrayIndexOutOfBoundsException e) {  // no arguments
        System.out.println("Syntax: CopyFile input output");
        System.exit(1);  // error code
    } catch(FileNotFoundException e) {  // unknown file
        System.out.println("File " + args[0] + " or file " +
          args[1] + " doesn't exist.\n");
        System.exit(2);  // another error
    } catch(IOException e) {  // another error, ie. disk full
        System.out.println(e.toString());
        System.exit(3);
    }
  }
}
```

Don't forget exception handling!

read(...) methods from InputStream

The method read() is abstract.
The methods
int read(byte[] b),
int read(byte[] b, int off, int len)
are non-abstract, but they internally invoke read().

Moral: any subclass of InputStream
must implement read().

read(b) == read(b, 0, b.length)

If b.length == 0 (or param len == 0 in the latter method),
then the methods return 0.
Nothing to read (EOF)? The methods return –1.

26

Combining stream functionalities

FileInputStream and FileOutputStream
can read / write only bytes.
Would be useful to have classes for
reading / writing int, longs, doubles...

Fortunately, there are classes
DataInputStream, DataOutputStream, but...
their constructors obtain an abstract stream
(InputStream, OutputStream, respectively).
How to work with files?

The solution is typical for Java I/O: combining filters.

```
FileInputStream fin = new FileInputStream("numbers.dat");
DataInputStream din = new DataInputStream(fin);
double x = din.readDouble();
```

27

Actually this is the Decorator design pattern

[http://www.javaworld.com/javaworld/jw-10-2001/jw-1012-designpatterns.html?page=5]

The Decorator pattern lets us add functionality to an object at runtime.

The idea. Object *x* from class X contains object *y* from Y.
Object *x* will be called a decorator.
x forwards method calls to *y*.
x conforms to the interface of *y*, which allows the decorator to be used as if *x* were an instance of Y.
(But typically *x* has its own capabilities, not existing in *y*.)

```
DataInputStream din = new DataInputStream(
        new BufferedInputStream(
        new FileInputStream("numbers.dat")));
```

28

FileOutputStream constructors

```
FileOutputStream(String name)
FileOutputStream(String name, boolean append)
FileOutputStream(File file)
FileOutputStream(File file, boolean append)
```

append == true → data are added
at the end of the file
(otherwise an existing file is deleted
and start from scratch)

29

Text I/O

Use FileReader, FileWriter classes.

FileReader: converts from the default codepage to Unicode. FileWriter: the other way around...

```
import java.io.*;
public class ReadByReader {
   public static void main(String[] args) {
      StringBuffer cont = new StringBuffer();
      try {
         FileReader inp = new FileReader(args[0]);
         int c;
         while ((c=inp.read()) != -1) cont.append((char)c);
         inp.close();
      } catch(Exception e) { System.exit(1); }
      String s = cont.toString();
      System.out.println(s);
   }
}
```

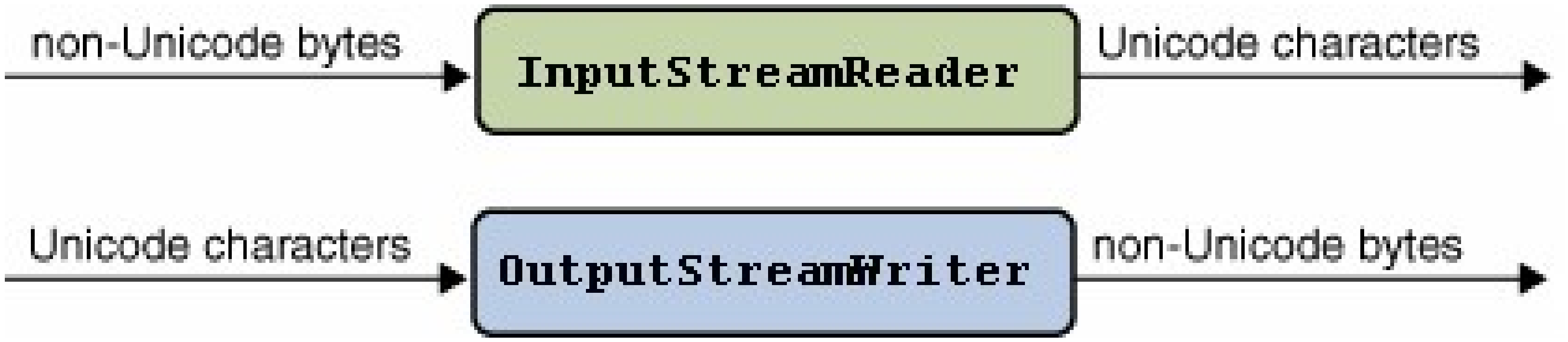
30

Non-standard character encoding

[http://download.oracle.com/javase/tutorial/i18n/text/stream.html]

If the characters in the file to read
adhere to non-standard encoding,
don’t use FileReader.
Use its parent class, InputStreamReader, instead.

OutputStreamWriter for writing, of course.



```
FileInputStream fis = new FileInputStream("test.txt");
InputStreamReader isr = new InputStreamReader(fis, "UTF8");
```

31

Reading / writing to / from a text (character) file

read() from InputStreamReader is a concrete method and it reads a single char (a variant reading a number of chars into an array also exists).

For a greater flexibility, use Scanner.

For writing to a text file, use PrintWriter.

Contains methods print, println, printf, format…

32

Object I/O

[http://www.cs.stir.ac.uk/courses/31V4/lectures/java5.pdf]

In Java, there exists a powerful mechanism for storing arbitrary objects in files (e.g., on disk). Or, more generally, sending them to a stream.

We save objects and then can recover them. This uses object serialization.

Basically, we can use methods writeObject() and readObject() – for any objects.

```
myOutFile.writeObject(thisObject);
val = (MyClass)myInFile.readObject(); // casting necessary
```

33

Object I/O, cont’d

We shall use java.io classes derived from OutputStream and InputStream:

- **OutputStream**
  - **FileOutputStream**
  - **ObjectOutputStream**
- **InputStream**
  - **FileInputStream**
  - **ObjectInputStream**

34

Object I/O, getting started...

[http://www.cs.stir.ac.uk/courses/31V4/lectures/java5.pdf]

For output:

```
ObjectOutputStream out;
out = new ObjectOutputStream (
          new FileOutputStream("myFile.dat") );
// use out.writeObject(...)
out.close();
```

For input (say, we read from the same file):

```
ObjectInputStream inp;
inp = new ObjectInputStream (
          new FileInputStream("myFile.dat") );
// use inp.readObject()
inp.close();
```

35

Object serialization

[http://www.cs.stir.ac.uk/courses/31V4/lectures/java5.pdf]

writeObject() is powerful.
It converts an object of any class into a bit sequence and then sends it to a stream,
in a way that it can be subsequently retrieved
(using **readObject()**) as an object of the original class.

Not surprisingly, static fields are NOT written to the stream.

Serialized objects may contain e.g.
other objects as their fields.

No problem with e.g. arrays either.
Even multi-dim arrays (btw, in Java a 2-d array is, formally speaking, an array of arrays).

36

How to serialize our own class

[http://www.cs.stir.ac.uk/courses/31V4/lectures/java5.pdf]

Our class must implement the interface Serializable.

public class Tank implements Serializable { ... }

Serializable is empty (=has no methods)!
Marker interfaces simply make possible to check
a data type, e.g. with instanceof operator.

That's what the writeObject(...) method does:
checks if
x instanceof Serializable == true.

If so, it sends the object to a stream, otherwise it doesn't.

(Other marker interfaces: Cloneable, RandomAccess...)

37

Remember about casts and checked exceptions

When reading a serialized object, it must be cast from an Object to the appropriate type.

Cast examples:

```
Student s = (Student)inpFile.readObject();

int[] tab = (int[])inpFile.readObject();
```

Don’t forget exceptions. The signatures are:

```
public final readObject()
  throws IOException, ClassNotFoundException;

public final void writeObject(Object obj)
        throws IOException;
```

38

The serialization API is not too good

(based on [J. Bloch, *Effective Java, 3rd ed.*, Item 41])

In class ObjectOutputStream:

```
public final void writeObject(Object obj)
        throws IOException
```

The passed argument should be of type Serializable, not Object.
Then we would have
compile-time errors, instead of runtime errors,
when a non-serializable object is passed.

What if an object is shared by a couple others as part of their state?

```
public class GradStudent
{ private Supervisor sv; ... }
// many grad students may have the same supervisor
```

No problem! Each GradStudent stores a ‘reference’ to the same Supervisor.

BUT! Those refs are not just mem addresses! Instead, SERIALizable objects get unique SERIAL numbers.

40

Serialization algorithm

- each encountered object reference gets a unique serial number,
- if an object reference is encounted for the first time, the object data are saved to the output stream,
- else (object ref already saved), it is written that here we have the same object as the one already saved with serial number x.

Reading back: reversed procedure.

41

Can we serialize selected fields?

Yes.

Mark a field with keyword *transient*
and it won’t be serialized.

E.g. (example from *Core Java, vol. 2*):

```
public class LabeledPoint implements Serializable
{
   . . .
   private String label;
   private transient Point2D.Double point;
}

// because Point2D.Double from java.awt.geom
// is not serializable and we want to avoid NonSerializableException
```

42

Serialize on our own

Sometimes we're not happy with the default serialization mechanism.

Cont'd previous slide example:
we do want to serialize the data from the Point2D.Double field.
Shall we write the serialization routine from scratch?

First we label the Point2D.Double field as transient (of course).
Then we implement

```
private void readObject(ObjectInputStream in)
  throws IOException, ClassNotFoundException;
```

```
private void writeObject(ObjectOutputStream out)
  throws IOException;
```

A serializable class is allowed to define those methods.

43

Serialize on our own, cont'd
[Core Java, vol. 2]

```
private void writeObject(ObjectOutputStream out)
   throws IOException
{
   out.defaultWriteObject();  // can be called only from writeObject(...) !
   out.writeDouble(point.getX());
   out.writeDouble(point.getY());
}

private void readObject(ObjectInputStream in)
   throws IOException
{
   in.defaultReadObject(); // can be called only from readObject(...) !
   double x = in.readDouble();
   double y = in.readDouble();
   point = new Point2D.Double(x, y);
}
```

44

Versioning

writeObject(...) writes several things, incl. class name and 8-byte class fingerprint.
The fingerprint is to ensure the class definition has not changed (between serialization and deserialization).

What if it has changed?

Indicate that the new class is compatible with the old one.
Run *serialver OurClass* for the old class to obtain its fingerprint.
Then add as (e.g.):

```
public static final long serialVersionUID =
-1814239825517340645L;
```

to the new version of OurClass.

No problem
if only methods
changed.

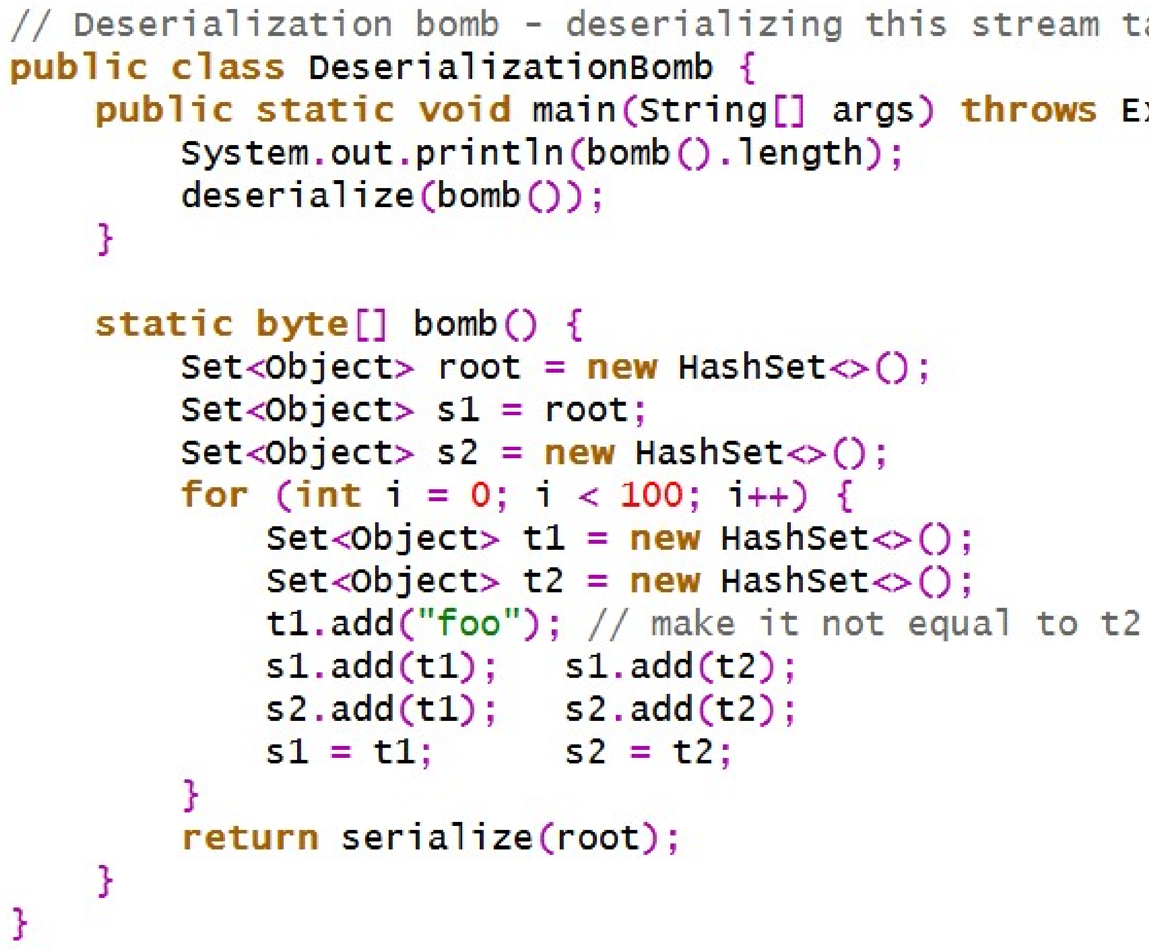
If new fields added:
risky.

If field type changed:
even worse.

45

Deserialization, beware! (J. Bloch, *Effective Java, 3rd Ed.*, Item 85)

A denial-of-service attack by causing the deserialization of a short stream that requires a long time to deserialize.

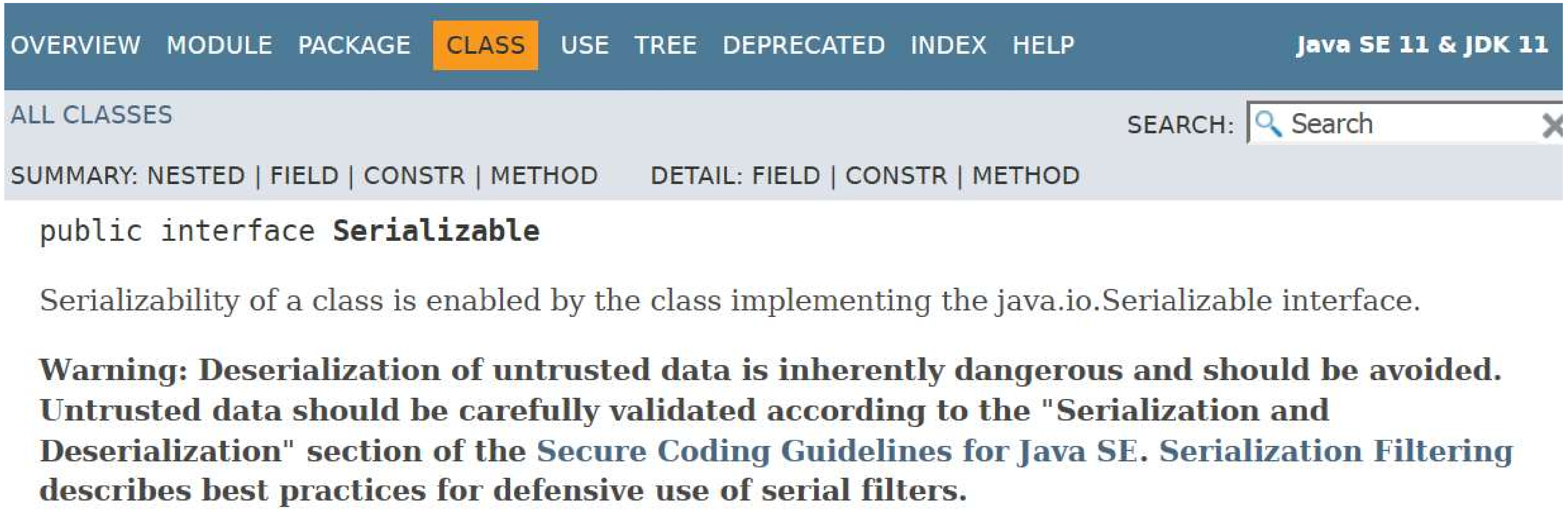


```java
// Deserialization bomb - deserializing this stream takes forever
public class DeserializationBomb {
    public static void main(String[] args) throws Exception {
        System.out.println(bomb().length);
        deserialize(bomb());
    }

    static byte[] bomb() {
        Set<Object> root = new HashSet<>();
        Set<Object> s1 = root;
        Set<Object> s2 = new HashSet<>();
        for (int i = 0; i < 100; i++) {
            Set<Object> t1 = new HashSet<>();
            Set<Object> t2 = new HashSet<>();
            t1.add("foo"); // make it not equal to t2
            s1.add(t1);   s1.add(t2);
            s2.add(t1);   s2.add(t2);
            s1 = t1;      s2 = t2;
        }
        return serialize(root);
    }
}
```

Deserializing a HashSet requires computing the hash codes of its elements. The 2 elements of root (which is a HashSet) are themselves hash sets containing 2 hash-set elements, each of which contains 2 hash-set elements, and so on, 100 levels deep (→ hashCode method called >2^{100} times).

Deserialization, beware!


OVERVIEW MODULE PACKAGE CLASS USE TREE DEPRECATED INDEX HELP
Java SE 11 & JDK 11
ALL CLASSES
SEARCH: Search
SUMMARY: NESTED | FIELD | CONSTR | METHOD
DETAIL: FIELD | CONSTR | METHOD

```
public interface Serializable
```

Serializability of a class is enabled by the class implementing the java.io.Serializable interface.

Warning: Deserialization of untrusted data is inherently dangerous and should be avoided. Untrusted data should be carefully validated according to the "Serialization and Deserialization" section of the Secure Coding Guidelines for Java SE. Serialization Filtering describes best practices for defensive use of serial filters.

(Some) alternatives to Java serialization:

- JSON (textual/inefficient/weakly typed)
- Google Protobuf

java.util.zip (compressed I/O)

The package java.util.zip contains several classes for data compression / decompression.

A few class examples:

Deflater – in-memory compression using zlib library

Inflater – in-memory decompression using zlib library

GzipOutputStream – compresses data using the GZIP format. To use it, construct a GZIPOutputStream that wraps a regular output stream and use write() to write compressed data

GzipInputStream – like above, but for reading data

ZipOutputStream, ZipInputStream – I won't offend your intelligence

ZipEntry – represents a single element (entry) in a zip file

48

Compression example (using gzip format)

[http://www.cafeaulait.org/slides/intljava/2001ny/javaio/44.html]

```
import java.io.*;
import java.util.zip.*;
import com.macfaq.io.*;


public class GZipper {

  public final static String GZIP_SUFFIX = ".gz";

  public static void main(String[] args) {

    for (int i = 0; i < args.length; i++) {
      try {
        FileInputStream in = new FileInputStream(args[i]);
        FileOutputStream fout = new FileOutputStream(args[i] + GZIP_SUFFIX);
        GZIPOutputStream out = new GZIPOutputStream(fout);
        byte[] buffer = new byte[256];
        while (true) {
          int bytesRead = in.read(buffer);
          if (bytesRead == -1) break;
          out.write(buffer, 0, bytesRead);
        }
        out.close();
      }
      catch (IOException e) {
        System.err.println(e);
      }
    }

  }

}
```

Writing to a stream of GZipOutStream type.
Which means it'll be compressed on-the-fly.

49

ZipFile class

[http://www.supinfo-projects.com/fr/2005/]

Used to read the entries in a zip file.
In order to create or to read these entries,
you have to pass a file as argument to the constructor:
ZipFile zipfile = new ZipFile(new File("c:\\test.zip"));
This instantiation can throw a ZipException,
as well as IOException.

```
try {
 zipfile = new ZipFile(new File("c:\\test.zip"));
} catch (ZipException e) { }
  catch (IOException e) { }
```

Now, files in a zip can be e.g. listed.

50

Listing a zip file

[http://javafaq.nu/java-example-code-225.html]

```
import java.util.zip.ZipFile;
import java.util.zip.ZipEntry;
import java.io.IOException;
import java.util.Enumeration;

public class ListZipFiles {
    private static void doListFiles(String zipFileName) {
        try {
            System.out.println("Opening zip file " + zipFileName);
            ZipFile zf = new ZipFile(zipFileName);
            int counter = 0;
            for (Enumeration entries = zf.entries(); entries.hasMoreElements();) {
                counter++;
                String zipEntryName = ((ZipEntry)entries.nextElement()).getName();
                System.out.println("Entry " + counter + " : " + zipEntryName);
            }
        } catch (IOException e) { e.printStackTrace(); System.exit(1); }
    }  // end of doListFiles

    public static void main(String[] args) {
        if (args.length != 1) {   // checking arg-list length
            System.err.println("Usage: java ListZipFiles zipfilename");
        } else { doListFiles(args[0]); }
    }  // end of main
}
```

51

java.io.File

Representation of a file or directory path name.
This class contains several methods for
working with the path name, deleting and renaming files,
creating new directories, listing the contents of a directory, etc.

```
File f1 = new File("/usr/local/bin/dog");
File f2 = new File("cat.txt");
File f3 = new File("c:/windows/system", "rabbit.gif");
```

You shouldn’t use File in new code (→ use Path).
No full support for modern file systems,
its API is inconsistent.
Examples: delete (and some other methods) don't
throw exceptions when fail (→ no useful error message),
little/inefficient support for metadata.

52

Absolute and relative paths

Absolute pathnames start with the root directory symbol, relative ones don’t.
Relative paths are resolved with respect to
user’s directory.
This directory is identified by system property user.dir.

In java.lang.System:

```
public static String getProperty(String key, String def)
public static String setProperty(String key, String value)
```

The cwd of the process:

```
Paths.get(".").toAbsolutePath().normalize().toString();
```

java.io.RandomAccessFile class

Supports read / write to a random access file.
A random access file behaves like a large byte array.
There is a kind of cursor (file pointer) into the implied array.
Input operations read bytes starting at the file pointer
and advance the file pointer past the bytes read.

Two constructors:
RandomAccessFile(File file, String mode)
RandomAccessFile(String pathname, String mode)

mode = "r" | "rw" | "rwd" | "rws"
(otherwise IllegalArgumentException)
"rwd" – like "rw", but each update to file’s content will be
written synchronously to the underlying storage device.
"rws" – like "rwd", but concerns update to file’s content or metadata.

File I/O in Java 7

java.nio.file,
java.nio.file.attribute packages

For example, use Path instead of File.

Path is an interface. To create an instance,
use static Path get(URI uri).
(or get(String first, String … more)).

Delete a file: Files.delete(Path path)
(throws IOException).

55

Walking the file tree (Java 7)

[http://docs.oracle.com/javase/tutorial/essential/io/walk.html]

A tree in a filesystem has a recursive nature. We may want to visit all the files in a file tree (to calculate their total size, find the last accessed file, delete all .class files, etc.).

In Java 7, there's FileVisitor interface for that, and two walkFileTree methods in Files class.

The simpler walkFileTree:
walkFileTree(Path, FileVisitor).

In FileVisitor interface there are 4 methods:

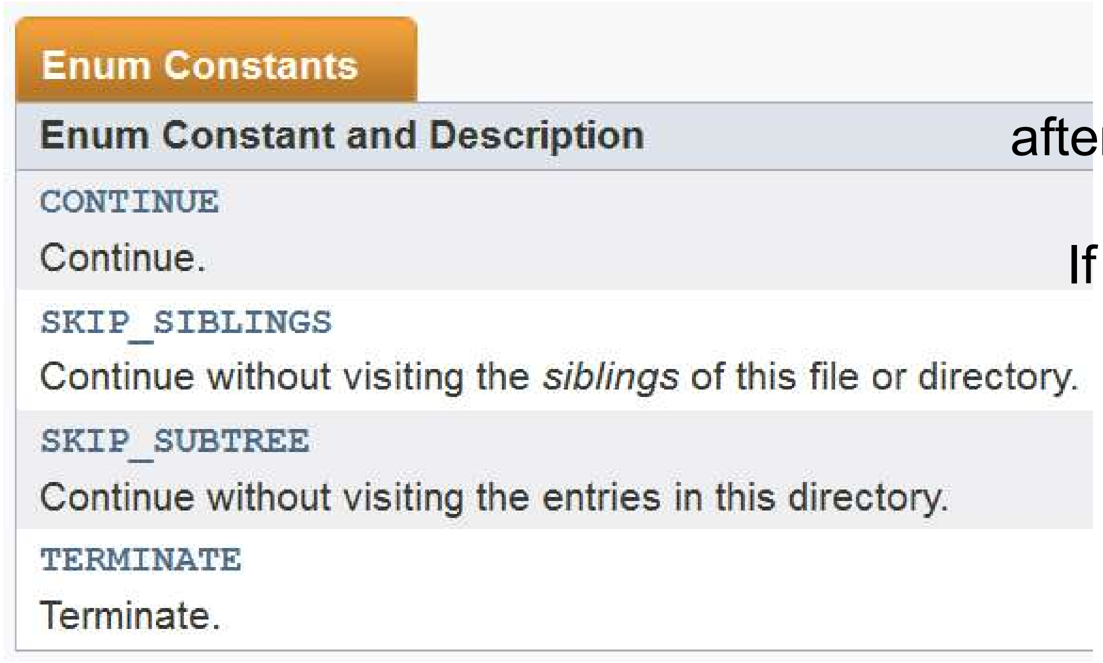
- preVisitDirectory,
- postVisitDirectory,
- visitFile,
- visitFileFailed.

Some methods from FileVisitor

[http://docs.oracle.com/javase/tutorial/essential/io/walk.html]

public FileVisitResult postVisitDirectory(Path dir, IOException exc)
public FileVisitResult visitFile(Path file, BasicFileAttributes attr)

FileVisitResult is an enum (extends Enum<FileVisitResult>).
Its constants:



Enum Constants

Enum Constant and Description
`CONTINUE` Continue.
`SKIP_SIBLINGS` Continue without visiting the *siblings* of this file or directory.
`SKIP_SUBTREE` Continue without visiting the entries in this directory.
`TERMINATE` Terminate.

postVisitDirectory – invoked after all the entries in a directory are visited.
If any errors are encountered, the specific exception is passed to the method.

Reflection

[ftp://ftp.oreilly.com/pub/conference/java2001/Portwood_Reflection.ppt]

Reflection (...) refers to the ability to observe and/or manipulate the inner workings of the environment programmatically.

Reflection in Java is possible thanks to
late dynamic binding.
Class loading occurs when the class is first referenced.

58

How the Class object works

[ftp://ftp.oreilly.com/pub/conference/java2001/Portwood_Reflection.ppt]

- Every class loaded into the JVM has a Class object
  - Corresponds to a .class file
  - The ClassLoader is responsible for finding and loading the class into the JVM
- At object instantiation…
  - The JVM checks to see if the class is already loaded into the virtual machine
  - Locates and loads the class if necessary
  - Once loaded, the JVM uses the loaded class to instantiate an instance

59

Reflection – what can I do with it?

[ftp://ftp.oreilly.com/pub/conference/java2001/Portwood_Reflection.ppt]

- Load a class
- Determine if it is a class or interface
- Determine its superclass and implemented interfaces
- Instantiate a new instance of a class
- Determine class and instance methods
- Invoke class and instance methods
- Determine and possibly manipulate fields
- Determine the modifiers for fields, methods, classes, and interfaces
- ...

Classes:

Class, Method, Field, Constructor, Modifier…

(most of them in java.lang.reflect package)

60

A very simple class
[http://www.onjava.com/pub/a/onjava/2007/03/15/
reflections-on-java-reflection.html]

```
public class Employee {
   public String _firstName;
   public String _lastName;
   private int _salary;

   public Employee()
   { this( "John", "Smith", 50000); }

   public Employee(String fn, String ln, int salary)
   { _firstName = fn;  _lastName = ln; _salary = salary; }

   public int getSalary() { return _salary; }

   public void setSalary(int salary) { _salary = salary; }

   public String toString() {
      return "Employee: " + _firstName +  " "
             + _lastName + " " + _salary;
   }
}
```

61

Looking into Employee's internals...

```
import java.lang.reflect.*;  // contains Modifier class

public class GetClassExample {
   public static void main(String[] args) {
      Employee employee = new Employee();
      Class klass = employee.getClass();
      System.out.println( "Class name: " + klass.getName());
      System.out.println(
            "Class super class: " + klass.getSuperclass());
      int mods = klass.getModifiers();
      System.out.println(
            "Class is public: " + Modifier.isPublic(mods));
      System.out.println(
            "Class is final: " +  Modifier.isFinal(mods));
      System.out.println(
            "Class is abstract: " + Modifier.isAbstract(mods));
   }
}
```

Output

```
Class name: Employee
Class super class: class java.lang.Object
Class is public: true
Class is final: false
Class is abstract: false
```

62

There are 3 ways to get hold of a class object

1. Calling *getClass* on an instance
   (we used it in the example).

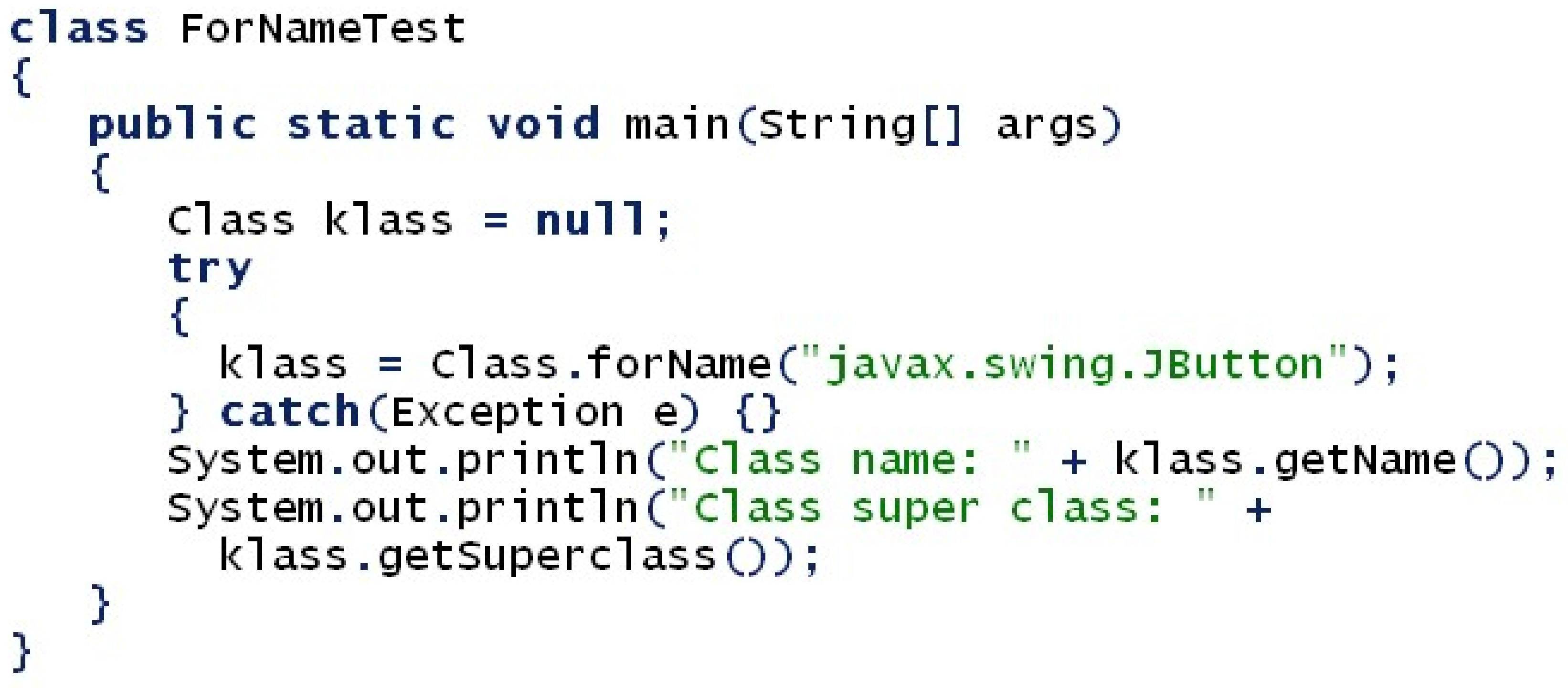
2. Getting it directly from a class name:
```
Class klass = Employee.class;
```

3. Creating a Class object from a string:

```
Class klass = Class.forName("javax.swing.JButton");
...
System.out.println("Class name: " + klass.getName());
...
System.out.println("Class super class: " +
klass.getSuperclass());
```

63

Creating a Class object from a string



```
class ForNameTest
{
   public static void main(String[] args)
   {
      Class klass = null;
      try
      {
        klass = Class.forName("javax.swing.JButton");
      } catch(Exception e) {}
      System.out.println("Class name: " + klass.getName());
      System.out.println("Class super class: " +
        klass.getSuperclass());
   }
}
```

About *forName:* you need to supply the full, complete with the package, name of the class.

64

Do smth with a Class object!

Make a new instance of the “hold” class:
newInstance() method.

```
public static void main(String[] args) throws
        ClassNotFoundException,
        InstantiationException,
        IllegalAccessException
{
        Class klass = Class.forName(args[0]);
        Object theNewObject = klass.newInstance();
        System.out.println("Just made: " + theNewObject);
}
```

Run with *Employee* argument (remember about the package name, if not the same); you will see:

```
Just made: Employee: John Smith 50000
```

65

Delaying the way an action is handling until the run-time ([Barteczko'00]) 1/2

```java
import static java.lang.System.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;

public class Reflection01 extends Frame {
  static void exit(String s) { out.println(s); System.exit(1); }
  public static void main(String[] args) { new Reflection01(args[0]); }

  Reflection01(String actionClassName) {
    Class actionClass = null;
    MouseListener m = null;
    try {
      actionClass = Class.forName(actionClassName);
      m = (MouseListener) actionClass.newInstance(); // Object - needs a cast
    } catch (Exception exc)
      { exit("Cannot create an object to handle the action."); }
    Button b = new Button("Action");
    b.addMouseListener(m);
    add(b);
    pack();
    show();
  }
}
```

66

Delaying the way an action is handling until the run-time ([Barteczko'00]) 2/2

```java
class JustReportOnConsole extends MouseAdapter {
  public void mouseReleased(MouseEvent e) {
    out.println(e.toString() + " on the component " +
      e.getComponent().toString());
  }
}
```

```
OUTPUT:
C:\jsources>java Reflection01 JustReportOnConsole

java.awt.event.MouseEvent[MOUSE_RELEASED,(107,8),button=1,modifiers=Button1,clic
kCount=1] on button0 on the component java.awt.Button[button0,4,29,136x23,label=
Action]
java.awt.event.MouseEvent[MOUSE_RELEASED,(107,8),button=1,modifiers=Button1,clic
kCount=1] on button0 on the component java.awt.Button[button0,4,29,136x23,label=
Action]
```

67

Reflection can simplify code.
A factory design pattern example

[ftp://ftp.oreilly.com/pub/conference/java2001/Portwood_Reflection.ppt]

Without reflection

```
public static Shape getFactoryShape(String s) {
   Shape temp = null;
   if (s.equals("Circle"))
      temp = new Circle();
   else
      if (s.equals("Square"))
         temp = new Square();
      else
         // ...
   return temp;
}
```

With reflection

```
try {
    temp = (Shape) Class.forName(s).
       newInstance();
}
catch (Exception e) { … }
```

68

Reflecting arrays

[http://www-128.ibm.com/developerworks/java/library/j-dyn0603/]

In Java, arrays are objects.
Like all objects, they have classes.
If you have an array, you can get the class of that array using the standard getClass method, just as with any other object.

However, getting the class without an existing instance works differently than for other types of objects. ☹

Even after you have an array class, there's not much you can do with it directly – the constructor access provided by reflection for normal classes doesn't work for arrays, and arrays don't have any accessible fields.
(Only the base java.lang.Object methods are defined for array objects.)

Reflecting arrays, cont’d

[http://www-128.ibm.com/developerworks/java/library/j-dyn0603/]

The special handling of arrays uses a collection
of static methods provided by the
java.lang.reflect.Array class.
The methods in this class let you create new arrays,
get the length of an array object,
and read and write indexed values of an array object.

Next slide:
a useful method for effectively resizing an existing array.
It uses reflection to create a new array of the same type,
then copies all the data across from the old array
before returning the new array.

70

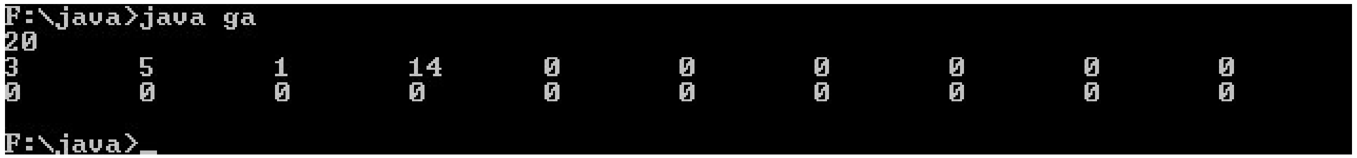
Reflecting arrays, cont’d

[http://www-128.ibm.com/developerworks/java/library/j-dyn0603/]

```java
import java.util.*;
import java.lang.reflect.*;

public class ga // grow array
{
  public static void main(String[] args)
  {
     int[] arr = {3,5,1,14};
     int[] new_arr = (int[])growArray(arr, 20);
     System.out.println(new_arr.length);
     for(int i: new_arr) System.out.print(i + "\t");
  }


  public static Object growArray(Object array, int size) {
    Class type = array.getClass().getComponentType();
    Object grown = Array.newInstance(type, size);
    System.arraycopy(array, 0, grown, 0,
        Math.min(Array.getLength(array), size));
    return grown;
  }
}
```



```
F:\java>java ga
20
3       5       1       14      0       0       0       0       0       0
0       0       0       0       0       0       0       0       0       0

F:\java>_
```

71

More on reflecting arrays

[http://www.java2s.com/Code/JavaAPI/java.lang/ClassgetComponentType.htm]

```java
import java.lang.reflect.Array;

public class MainClass {
  public static void main (String args[]) {
    Object array = Array.newInstance(int.class, 3);

    Class type = array.getClass();
    if (type.isArray()) {
      Class elementType = type.getComponentType();
      System.out.println("Array of: " + elementType);
      System.out.println("Array size: " + Array.getLength(array));
    }

    String[] str_arr = { "Mary", "had", "a", "little", "lamb" };

    type = str_arr.getClass();
    if (type.isArray()) {
      Class elementType = type.getComponentType();
      System.out.println("Array of: " + elementType);
      System.out.println("Array size: " + Array.getLength(str_arr));
    }
  }
}
```

72

Generic type inference improvements in Java 8

```
List<String> stringList = new ArrayList<>();
stringList.add("A");
stringList.addAll(Arrays.asList());
```

Works in Java 8.

But not in Java 7!

error: no suitable method found for addAll(List<Object>) ...
method List.addAll(Collection<? extends String>) is not applicable
(actual argument List<Object> cannot be converted to
Collection<? extends String> by method invocation conversion)

Corrected for Java 7:

```
List<String> stringList = new ArrayList<>();
stringList.add("A");
stringList.addAll(Arrays.<String>asList());
```

File class, cont’d

Selected methods (1/2)

```
boolean canRead()       // Returns true if the file is readable
boolean exists()        // Returns true if the file exists
boolean isDirectory()   // Returns true if the file name is a directory
boolean isFile()        // Returns true if the file name
                   // is a "normal" file (depends on OS)
long length()           // Returns the file length
boolean setReadOnly() // (since 1.2) Marks the file read-only
                   // (returns true if succeeded)
boolean delete()        // Deletes the file specified by this name.
boolean mkdir()         // Creates this directory.
                   // All parent directories must already exist.
File[] listRoots()      // Returns the root directories of
                   // available filesystems (e.g. C:\, D:\, E:\ in Windows
                   // and only / in Unix
```

74

File class, cont’d

Selected methods (2/2)

```
String[] list()     // Returns an array of Strings with names of the
                    // files from this directory.  Returns null if not a dir.
String[] list(FileNameFilter filter)

File[] listFiles()
File[] listFiles(FileFilter)
File[] listFiles(FileNameFilter)  // (all three since 1.2)
```

FileFilter: public interface with only one method:

boolean accept(File pathname);

// tests if the specified pathname should be included in a pathname list

FileNameFilter: similar interface

but the arg list for accept() different

Beware: JFileChooser (Swing) makes use of javax.swing.filechooser.FileFilter, not java.io.FileFilter!

75