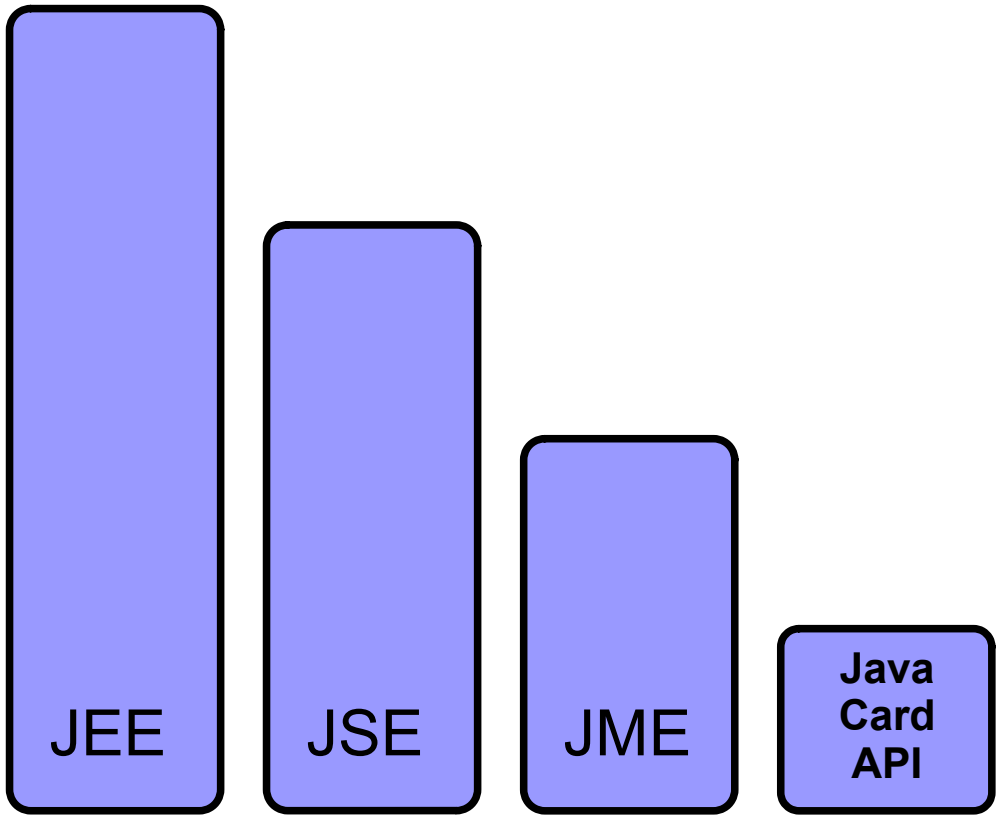
Platforms, etc.

History

- JDK 1.0 – 1996
- JDK 1.1 – 1997
  - Inner classes
- Java 2 platform – 2000
  - JDK 1.2, 1.3 – changes in libraries only
- JDK 1.4 – 2002
  - Assert
- JDK 5.0 – 2004
  - changes in the language
    - generics
    - annotations
    - . . .
- JDK 6 – 2006
- JDK 7 – 2011 – „small“ changes in the language
- JDK 8 – 2014 – lambdas,…
- JDK 9 – 2017 – modules
- JDK 10 – 2018 – local variables type inference (var)
- JDK 11 – 2018 – extended usage of var
  - std library reduction
  - long-term support
- JDK 12 – 2019 – extended switch (a “preview” feature)
- JDK 13 – 2019 – further switch modification, text blocks (still “preview”)

Java platform

- JSE – standard edition
- JEE – enterprise edition
- JME – micro edition



Performance

- originally (~ JDK 1.1, 1998)
  - Java programs 6 times slower than C
- now:
  - Just-In-Time (JIT) compilation
    - during launching the program is compiled to native code
    - native code is executed
    - slow start, then fast
- performance ~ comparable with native applications
- big memory consumption

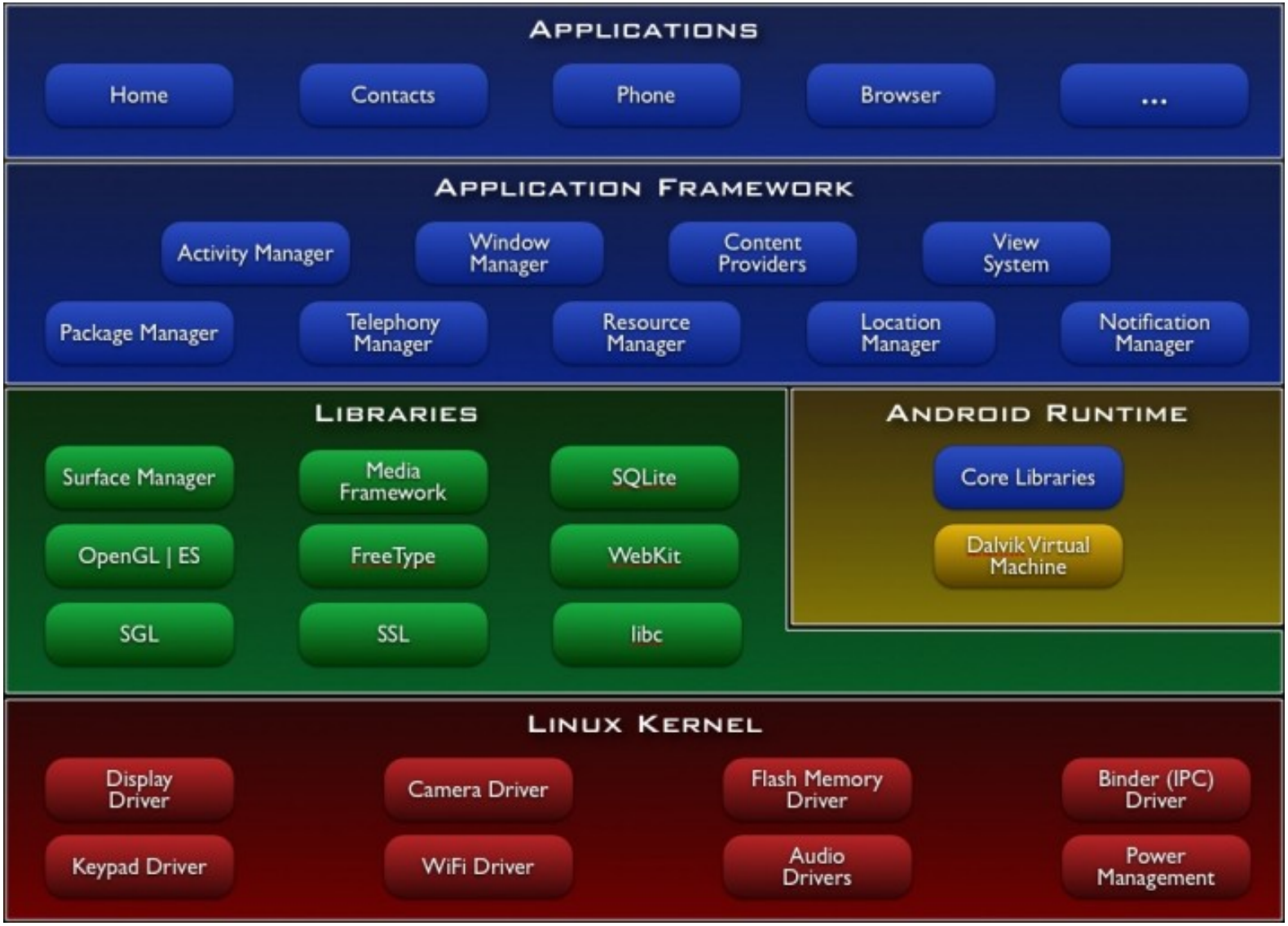
Java implementations

- Oracle (Sun Microsystems)
  - "official" implementation
  - Windows, Solaris, Linux, macOS
- OpenJDK
  - http://openjdk.java.net/
  - open-source
  - supported by Oracle (Sun Microsystems)
  - official implementation created from OpenJDK
- IBM
  - IBM JDK
  - 2017 => Eclipse OpenJ9 – open-source

Java implementations

- Jikes RVM
  - Research Virtual Machine
  - open-source
  - for testing extensions
  - written in Java
    - "self-hosting"
      - does need another JVM to run
    - boot-image writer
      - a Java program, which is executed in an existing JVM
    - boot-image loader
      - a program written in C++
  - does not support complete Java API

Android



source: http://developer.android.com/

Bck2brwsr

- http://wiki.apidesign.org/wiki/Bck2Brwsr
- Java running in a browser
- Project goals
  - „Create small Java capable to boot fast and run in 100% of modern browsers including those that have no special support for Java.
  - Demonstrate that Java has benefits over JavaScript when creating larger HTML5 applications“
  - ...

History and future

Changes in the language – Java 5

- static import
- auto-boxing and auto-unboxing
- new **for** cycle
- generics
- **enum**
- methods with variable number of parameters (printf)
- annotations (metadata)

Java 7

- changes
  - changes in syntax
  - support for dynamic languages (a new instruction in bytecode)
  - changes in NIO
  - Nimbus (Swing LaF)
  - new version of JDBC
  - ...

Java 7 – changes in syntax

- expressing constants
  - binary constants
    - `0b010101`
  - underscores in numerical literals
    - `1_000_000`
- String type in the switch

```
String month;
...
switch (month) {
  case "January":
  case "February":
  ...
}
```

Java 7 – changes in syntax

- operator **<>**
  - simplified instantiation of generics
  - type in <> is automatically inferred
  - ex.

```
List<String> list = new ArrayList<>();
List<List<String>> list = new ArrayList<>();
List<List<List<String>>> list =
                                  new ArrayList<>();
Map<String, Collection<String>> map =
                              new LinkedHashMap<>();
```

- question
  Why <> is necessary? I.e. why is not sufficient

```
List<String> list = new ArrayList();
```

Java 7 – changes in syntax

- the interface **`AutoClosable`** and extended **`try`**
  - ex:

```
class Foo implements AutoClosable {
   ...
   public void close() {  …  }
}

try ( Foo f1 = new Foo(); Foo f2 = new Foo() ) {
    ...
} catch (…) {
    ...
} finaly {
    ...
}
```

  - at the end of **`try`** (normally or by an exception), **`close()`** is always called on all the objects in the **`try`** declaration
    - called in the reverse order than declared

Java 7 – changes in syntax

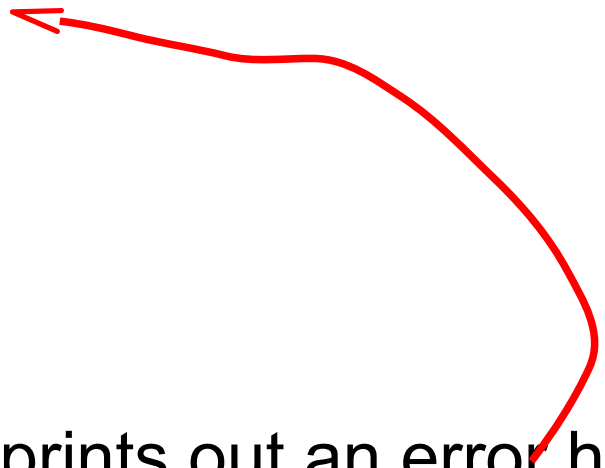
- multi- **`catch`** for several exceptions
  - ex:

```
try {
   ...
} catch (Exception1 | Exception2 ex) {
   ...
}
```

- better type control during re-**`throw`**

Is the following code correct

```
private void foo(int i) throws Ex1, Ex2 {
   try {
     if (i < 0) {
       throw new Ex1();
     } else {
       throw new Ex2();
     }
   } catch (Exception ex) {
     throw ex;
   }
}
```



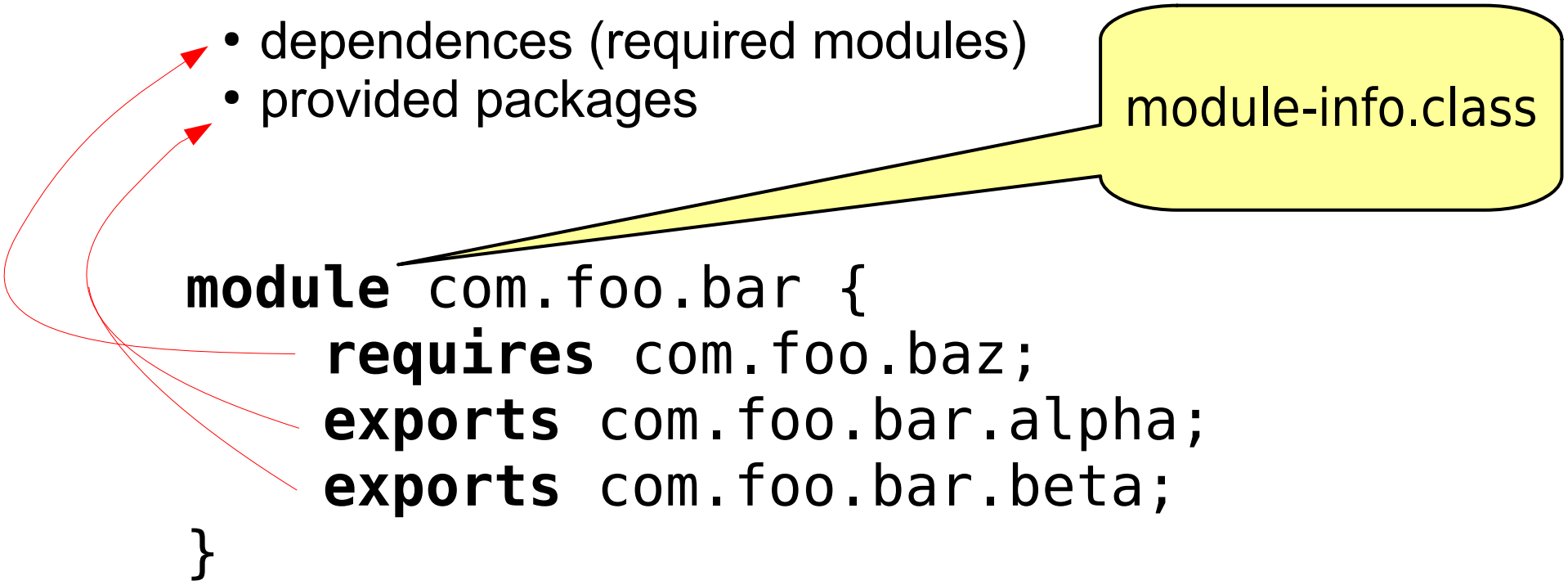
- in Java 7 **yes**
- in Java 6 no
  - the compiler prints out an error here

Java 8

- type annotations
  - type use can be annotated
  - repeating annotations
- default and static methods in interfaces
- lambda expressions
- generic type inference
- profiles
  - a “subset” of the std library
    - javac -profile ...

Java 9 – modules

- Module
  - named and self-describing collection of packages with types (classes,...) a data
  - declares
    - dependences (required modules)
    - provided packages



Java 9 – modules

- JSE platform – divided into a set of modules

```
module java.base {
    exports java.io;
    exports java.lang;
    exports java.lang.annotation;
    exports java.lang.invoke;
    exports java.lang.module;
    exports java.lang.ref;
    exports java.lang.reflect;
    exports java.math;
    exports java.net;
    ...
}
```

Java 10

- local variables type inference

```
var s = "hello";
var list = new ArrayList<String>();
```

- var – reserved type name
- it is not a keyword

- requires initialization

Java 11

- the var type-inferencing for lambda parameters

Java 13

- modified switch
  - arrow instead of colon
  - no need for break
  - multiple values
  - usage as expression
- text blocks

```
"""multiline
    string"""
```

Java

Generics

Introduction

- *similar* to the templates in C#/C++
  - **but only on first view**
- typed arguments
- goal
  - clear code
  - type safety

Motivational example

- without generics (<=Java 1.4)

```
List myIntList = new LinkedList();
myIntList.add(new Integer(0));
Integer x = (Integer)myIntList.iterator().next();
```

- >= Java 5

```
List<Integer> myIntList = new LinkedList<Integer>();
myIntList.add(new Integer(0));
Integer x = myIntList.iterator().next();
```

- no explicit casting
- type checks during compilation

Definition of generics

```
public interface List<E> {
  void add(E x);
  Iterator<E> iterator();
  E get(int i);
}
public interface Iterator<E> {
  E next();
  boolean hasNext();
}
```

- `List<Integer>` can be seen as

```
public interface IntegerList {
  void add(Integer x);
  Iterator<Integer> iterator();
}
```

- but in reality no such code exists
  - no code is generated as in C++

Type relations

- no changes in typed arguments are allowed

```
List<String> ls = new ArrayList<String>();
List<Object> lo = ls;
```

```
lo.add(new Object());
String s = ls.get(0);
```

error – assigning `Object` to `String`

- second line causes compilation error
- `List<String>` is not subtype of `List<Object>`

Type relations

- example – printing all elements in a collection

  <= Java 1.4

```
void printCollection(Collection c) {
  Iterator i = c.iterator();
  for (k = 0; k < c.size(); k++) {
   System.out.println(i.next());
  }
}
```

  naive attempt in Java 5

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

  - does not work (see the previous example)

Type relations

- `Collection<Object>` is not supertype of all collections
- correctly

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

- `Collection<?>` is supertype of all collections
  - collection of unknown
  - any collection can be assigned there
- BUT – to `Collection<?>` nothing can be added

```
Collection<?> c = new ArrayList<String>();
c.add(new Object());
```

<= compilation error

- `get()` can be called – return type is `Object`

Type relations

- ? - wildcard
- bounded wildcard

```
public abstract class Shape {
  public abstract void draw(Canvas c);
}
public class Circle extends Shape { ... }
public class Canvas {
  public void drawAll(List<Shape> shapes) {
    for (Shape s:shapes) {
      s.draw(this)
    }
}}
```

- can draw lists of the type `List<Shape>` only but not e.g. `List<Circle>`

Type relations

covariant

- solution – bounded ?

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

- but still you cannot add to this List

```
shapes.add(0, new Rectangle());
```

compilation error

Generic methods

```
static void fromArrayToCollection(Object[] a,
  Collection<?> c) {
  for (Object o : a) {
    c.add(o);
  }
}
```

← compilation error

```
static <T> void fromArrayToCollection(T[] a,
  Collection<T> c) {
  for (T o : a) {
    c.add(o);
  }
}
```

← OK

Generic methods

- usage
  - the compiler determines actual types automatically

```
Object[] oa = new Object[100];
Collection<Object> co = new ArrayList<Object>();
fromArrayToCollection(oa, co); // T → Object
String[] sa = new String[100];
Collection<String> cs = new ArrayList<String>();
fromArrayToCollection(sa, cs); // T → String
fromArrayToCollection(sa, co); // T → Object
```

- bounds can be used with methods also

```
class Collections {
  public static <T> void copy(List<T> dest, List<?
  extends T> src){...}
}
```

Type inference

- compiler cannot always determine the type
  - example

```
class Collections {
  static <T> List<T> emptyList();
  …
}
```

  - `List<String> listOne = Collections.emptyList();`
    - OK
  - `void processStringList(List<String> s) {`

```
    ..
}
processStringList(Collections.emptyList());
```

    - cannot by compiled (in Java 7)

Type inference

- we can provide “help” to the compiler
  - processStringList(Collections.<String>emptyList());

- since Java 8 the example can be compiled without the “help”
  - better type inference

Generic methods and ?

- when use generic methods and when wildcards

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

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

- What is better?

Generic methods and ?

- when use generic methods and when wildcards

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

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

- generic methods – relations among several types

Generic methods and ?

- it is possible to use both generic methods and wildcards together

```
class Collections {
  public static <T> void copy(List<T> dest,
                              List<? extends T> src) {....}
}
```

- it can be also written as

```
class Collections {
  public static <T, S extends T>
         void copy(List<T> dest, List<S> src) {....}
}
```

- first variant is “correct”

Array and generics

- array of generics
  - can be declared
  - cannot be instantiated

```
List<String>[] lsa = new List<String>[10];   wrong
List<?>[] lsa = new List<?>[10];  OK + warning
```

- why? arrays can be cast to Object

```
List<String>[] lsa = new List<String>[10];
Object[] oa = lsa;
List<Integer> li = new ArrayList<Integer>();
li.add(new Integer(3));
oa[1] = li;
String s = lsa[1].get(0);   ClassCastException
```

“Old” and “new” code

- “old” code without generics

```
public class Foo {
  public void add(List lst) { ... }
  public List get() { ... }
}
```

- “new” code that uses the “old” one

```
List<String> lst1 = new ArrayList<String>();
Foo o = new Foo();
o.add(lst1);    ← OK - List corresponds to  List<?>
List<String> lst2 = o.get(); ← compilation warning
```

“Old” and “new” code

- “new” code with generics

```
public class Foo {
  public void add(List<String> lst) { ... }
  public List<String> get() { ... }
}
```

- “old” code that uses the “new” one

```
List lst1 = new ArrayList();
Foo o = new Foo();
o.add(lst1);    ← compilation warning
List lst2 = o.get(); ← OK - List  corresponds to  List<?>
```

"Erasure"

```
public String loophole(Integer x) {
  List<String> ys = new LinkedList<String>();
  List xs = ys;
  xs.add(x);              ← warning
  return ys.iterator().next();
}
```

- at runtime, it behaves as

```
public String loophole(Integer x) {
  List ys = new LinkedList();
  List xs = ys;
  xs.add(x);
  return (String)ys.iterator().next();   ← runtime error
}
```

"Erasure"

- during compilation, all information about generic types are erased
  - "erasure"

  - type parameters are erased (List<Integer> → List)
  - type variables are replaced by the most common type
  - casts added

Code of generic classes

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

- What is printed out?
  a) true
  b) false

Code of generic classes

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

- What is printed out?
  a) true
  b) false

Casts, instanceof

```
Collection cs = new ArrayList<String>();
if (cs instanceof Collection<String>)....
```

- impossible

```
Collection<String> cstr = (Collection<String>) cs;
```

- warning
- cannot be obtain at runtime

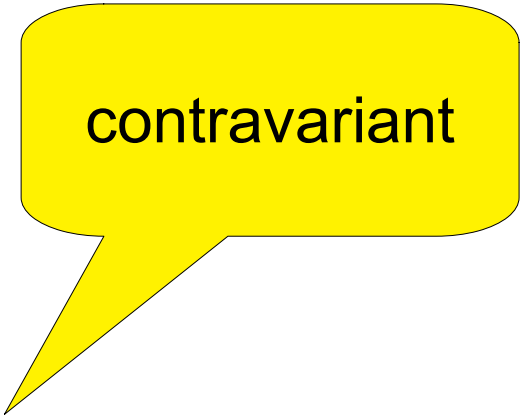
```
<T> T badCast(T t, Object o) {return (T) o;}
```

- warning

```
<T> T[] makeArray(T t) {
  return new T[100];        ← impossible
}
```

Additional type relations

```
class Collections {
  public static <T> void copy(List<T> dest, List<?
  extends T> src){...}
}
```



- actual declaration is

```
class Collections {
  public static <T> void copy(List<? super T> dest,
  List<? extends T> src){...}
}
```

- it is possible to add to the **<? super T>** collection

Additional type relations

- super can be used with gen. methods only
- cannot be used with gen. types
  - would not bring anything

```
class Foo<T super Number > {
    private T v;
    public Foo(T t) { v = t; }
}
```

  - after *erasure*

```
class Foo {
    private Object v;
    public Foo(Object t) { v = t; }
}
```

  - it would only guarantee that as a parameter a supertype of Number can be used
  - it would not guarantee that in the variable is always an instance of a supertype of Number

Converting "old" code to new

```
interface Comparator<T>
  int compare(T fst, T snd);
}

class TreeSet<E> {
  TreeSet(Comparator<E> c)
  ...
}
```

- `TreeSet<String>`
  - it is possible to use both Comparator<String> and Comparator<Object>

→
```
class TreeSet<E> {
  TreeSet(Comparator<? super E> c)
  ...
}
```

Converting "old" code to new

```
Collections {
  public static <T extends Comparable<T>>
                          T max(Collection<T> coll);
}
```

```
class Foo implements Comparable<Object> {...}
Collection<Foo> cf = ...
Collections.max(cf)
```

does not work

- correctly

```
public static <T extends Comparable<? super T>>
                          T max(Collection<T> coll);
```

Converting "old" code to new

```
public static <T extends Comparable<? super T>>
                                T max(Collection<T> coll);
```

- erasure
  - public static Comparable max(Collection coll)
  - is not compatible with the “old” metod max
    - public static Object max(Collection coll)
- more correctly

```
public static <T extends Object & Comparable<? super
  T>> T max(Collection<T> coll);
```

  - several type can be specified: T1 & T2 & ... & Tn
  - "erasure" takes the first one

- fully correctly

```
public static <T extends Object & Comparable<? super
  T>> T max(Collection<? extends T> coll);
```

Java

Annotations

Overview

- annotations ~ metadata
  - “data about data”
  - additional information about a part of code, which does not (directly) influence program functionality
- since JDK 5
- examples
  - @Deprecated
  - @SuppressWarnings
  - @Override

Motivation for annotations

- in fact, annotations have existed before JDK 5
  - but were not defined systematically, and
  - could not be added (easily)
  - e.g.:
    - the modifier **transient**
    - @deprecated element in a javadoc comment
    - …
- XDoclet
  - http://xdoclet.sourceforge.net/
  - adding annotations to “old” Java
  - as definable tags in javadoc comments
  - anything can be generated from them
    - contains many predefined tags and transformations
  - originally, it was a tool supporting development of EJB components

Usage

- annotations can be used in fact to any element of a program
  - classes
  - interfaces
  - fields
  - methods
  - constructors
  - packages
  - type usage (since Java 8)
- general rule
  annotation can be used on places, where modifiers can be used
  - exception – annotations for packages (written to the special file **`package-info.java`**) and
  - type usage
- an annotation usage can be restricted

Usage

- e.g.:

```
class A {
    @Override public boolean equals(A a) { ... }
    @Deprecated public void myDeprecatedMethod() {
      ...
    }
 }
```

- annotations can have parameters

```
@SuppressWarnings("unchecked") public void foo() {
```

Usage

- annotation of type use (Java 8)
  - `new @Interned MyObject();`
  - `myString = (@NonNull String) str;`
  - `class UnmodifiableList<T> implements @Readonly List<@Readonly T> { ... }`
  - `void monitorTemperature() throws @Critical TemperatureException { ... }`

Usage

- can be used among modifiers in any order
  - common usage – first annotations, then modifiers
- any number of annotations can be used to a single element

- Java 5-7 – a single annotation **cannot** be applied several times to a single element
  - even if used with different parameters
- Java 8+ – a single annotation **can** be applied several times to a single element

Definition

- similarly to interfaces
  - @interface
  - methods without implementation

- e.g.

```
public @interface RequestForEnhancement {
   int  id();
   String synopsis();
   String engineer() default "[unassigned]";
   String date()     default "[unimplemented]";
}
```

Definition

- “special” annotations

- marker
  - no body
    - no parameters when annotation is used

  - `public @interface Preliminary { }`

- single-member
  - a single method named ***value***
    - any type
  - when used, only annotation and parameter value is written

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

Definition

- usage of the previous annotations

```
@RequestForEnhancement(
    id       = 2868724,
    synopsis = "Enable time-travel",
    engineer = "Mr. Peabody",
    date     = "4/1/3007"
)
public static void travelThroughTime(Date destination)
{ ... }

@Preliminary public class TimeTravel { ... }

@Copyright("2002 Yoyodyne Propulsion Systems")
public class OscillationOverthruster { ... }
```

Definitions

- same as for interfaces
  - place of declaration
  - scope validity
  - scope of visibility
- must not be a generic type
- must not contain extends
  - by default the extends **java.lang.annotation.Annotation**
- any number of methods
- annotation T must not contain a method returning T
  - directly but also indirectly

```
@interface SelfRef { SelfRef value(); }
@interface Ping { Pong value(); }
@interface Pong { Ping value(); }
```

Definition

- methods must not have any parameters
- methods must not be generic
- methods must not declare throws
- returning value must be either:
  - primitive type
  - String
  - Class
  - Enum
  - annotations
  - array of the types above

Definition

- when used, annotation must contain a tuple name-value for each method
  - does not hold for methods with default value
- values must not be **null**

Pre-defined annotations

- annotations for usage on annotations
  - restrict usage of the annotation
  - in package java.lang.annotation
- **@Target**
  - single-member
  - restricts applicability of the annotation
  - possible values (enum ElementType)
    - ANNOTATION_TYPE
    - CONSTRUCTOR
    - FIELD
    - LOCAL_VARIABLE
    - PACKAGE
    - METHOD
    - PARAMETER
    - TYPE – can be used on class, interface, enum, annotation
    - TYPE_PARAMETER – since Java 8
    - TYPE_USE – since Java 8
    - MODULE – since Java 9

Pre-defined annotations

- **@Retention**
  - single-member
  - defines when the annotation can be used
  - possible values (enum RetentionPolicy)
    - SOURCE – source code only
    - CLASS – at compiler time
    - RUNTIME – at run-time

```
@Retention(RetentionPolicy.RUNTIME)
public @interface Foo { }
```

Repeating annotations

- since Java 8

```
@Schedule(dayOfMonth="last")
@Schedule(dayOfWeek="Fri", hour="23")
public void foo() { ... }
```

- for compatibility reasons, repeating annotations are stored in an automatically generated container
  - the container has to be prepared

```
@Repeatable(Schedules.class)
public @interface Schedule { ... }

public @interface Schedules {
    Schedule[] value;
}
```

Obtaining annotations at runtime

- via Reflection API
- the interface `AnnotatedElement`
  - `isAnnotationPresent` – if an annotation present
  - `getAnnotation` – returns annotation of the given type, if it is applied
  - `getAnnotations` – returns all annotations
  - `getDeclaredAnnotations` – returns declared annotations (without inherited)

Processing SOURCE annotations

- annotation processors
  - specified to the compiler
  - the parameter **`-processor`**
  - **`javax.annotation.processing.Processor`**
  - since Java 6

- Annotation Processing Tool (APT)
  - an external tool for annotation processing
  - Java 5
  - since JDK 8 – APT and corresponding API marked as deprecated

Example – Unit Testing

```
import java.lang.annotation.*;
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Test { }



public class Foo {
    @Test public static void m1() {
        …
    }
    public static void m2() {
        ...
    }
    @Test public static void m3() {
        ...
    }
}
```

Example – Unit Testing

```
import java.lang.reflect.*;
public class RunTests {
   public static void main(String[] args)
                                          throws Exception {
      int passed = 0, failed = 0;
      for (Method m :
                Class.forName(args[0]).getMethods()) {
         if (m.isAnnotationPresent(Test.class)) {
            try {
               m.invoke(null);
               passed++;
            } catch (Throwable ex) {
               System.out.printf("Test %s
                        failed: %s %n", m, ex.getCause());
               failed++;
            }
         }
      }
      System.out.printf("Passed: %d,
                              Failed %d%n", passed, failed);
   }
}
```