JAVA

Assertions

Assertion

- since Java 1.4
- the statement with a **`boolean`** expression
- a developer supposes that the expression is always satisfied (evaluates to **true**)
- if it is evaluated to **false** -> error
- intended for debugging
  - assertions can be enabled or disable
    - for whole program or for several classes only
  - disabled by default
  - ***must not*** have any side effects
- javac 1.4 – code have to be compiled with the argument `-source 1.4`

Usage

```
assert Expression1;
assert Expression1 : Expression2;
```

- disabled assertions – the statement does nothing
  - expressions are not evaluated
- enabled assertions
  - Expression1 is **true** – program continues normally
  - Expression1 is **false**
    - Expression2 is presented
      `throw new AssertionError(Expression2)`
    - Expression2 is not presented
      `throw new AssertionError()`

Enabling and disabling

- arguments for the virtual machine
- enabling
  -ea[:PackageName...|:ClassName]
  -enableassertions[:PackageName...|:ClassName]
- disabling
  -da[:PackageName...|:ClassName]
  -disableassertions[:PackageName...|:ClassName]
- without class or package – for all classes
- assertions in "system" classes
  -esa | -enablesystemasserions
  -dsa | -disablesystemasserions

- decision whether the assertions are enabled, is evaluated just once during initialization of a class (before anything is called/used on this class)

java.lang.AssertionError

- extends `java.lang.Error`
- constructors

```
AssertionError()
AssertionError(boolean b)
AssertionError(char c)
AssertionError(double d)
AssertionError(float f)
AssertionError(int i)
AssertionError(long l)
AssertionError(Object o)
```

Examples

- invariants

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

Examples

- "unreachable places" in a program

```
class Directions {
  public static final int RIGHT = 1;
  public static final int LEFT = 2;
}
...
switch(direction) {
  case Directions.LEFT:
    ...
  case Directions.RIGHT:
    ...
  default:
    assert false;
}
```

Examples

- preconditions
  - testing arguments of `private` methods

```
private void setInterval(int i) {
  assert i>0 && i<=MAX_INTERVAL;
  ...
}
```

  - unrecommended for testing arguments of public methods

```
public void setInterval(int i) {
  if (i<=0 && i>MAX_INTERVAL)
    throw new IllegalArgumentException();
  ...
}
```

Examples

- postconditions

```
public String foo() {
   String ret;
   ...
   assert ret != null;
   return ret;
}
```

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++

New instances

```
ArrayList<Integer> list = new ArrayList<Integer>();
ArrayList<ArrayList<Integer>> list2 =
new ArrayList<ArrayList<Integer>>();
HashMap<String, ArrayList<ArrayList<Integer>>> h =
new HashMap<String, ArrayList<ArrayList<Integer>>>();
```

- since Java 7 (“diamond” operator)

```
ArrayList<Integer> list = new ArrayList<>();
ArrayList<ArrayList<Integer>> list2 =
new ArrayList<>();
HashMap<String, ArrayList<ArrayList<Integer>>> h =
new HashMap<>();
```

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

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

- 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){...}
}
```

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 o = lsa;
Object[] oa = (Object[]) o;
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<?>
```

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){...}
}
```

Java

Enum

Enumerations

- <= Java 1.4

```
public static final int COLOR_BLUE = 0;
public static final int COLOR_RED = 1;
public static final int COLOR_GREEN = 2;
```

- possible problems
  - type (un)safety
  - no namespace
  - constants hard-compiled in clients
  - only numbers when printed

Enum

```
public enum Color { BLUE, RED, GREEN }
...
public Color clr = Color.BLUE;
```

- “normal” class
  - can have fields, methods, even the main method
  - subclass of `java.lang.Enum`
  - for each value – single instance
    - public static final field
    - protected constructor

„Enum without enum“

- how to implement enum in Java 1.4
  - (and how enums are implemented)

```
class Color {
  private int value;

  public static final Color RED = new Color(0);
  public static final Color GREEN = new Color(1);
  public static final Color BLUE = new Color(2);

  protected Color(int v) {
    value = v;
  }
   ...
}
```

java.lang.Enum

```
public abstract class Enum <E extends
  Enum<E>> { ... }
```

- methods
  - `String name()`
  - `int ordinal()`

- each enum has the methods `values()`
  - returns an array with all enum's values

```
public Colors clr = Colors.BLUE;
System.out.println(clr);   →  BLUE
```

Fields and methods

```
public enum Planet {
  MERCURY (3.303e+23, 2.4397e6),
  VENUS (4.869e+24, 6.0518e6),
  EARTH (5.976e+24, 6.37814e6),
  ...

  private final double mass;
  private final double radius;

  Planet(double mass, double radius){
    this.mass = mass;
    this.radius = radius;
  }

  double surfaceGravity() {
    return G * mass / (radius * radius);
  }
}
```

Fields and methods

- example

```
public enum Operation {
  PLUS, MINUS, TIMES, DIVIDE;

  double eval(double x, double y){
    switch(this) {
      case PLUS:   return x + y;
      case MINUS:  return x - y;
      case TIMES:  return x * y;
      case DIVIDE: return x / y;
    }
    throw new AssertionError("Unknown op: " + this);
  }
}
```

Fields and methods

- abstract methods
- particular implementations with each of the values

```
public enum Operation {
  PLUS { double eval(double x, double y) { return x+y; }},
  MINUS { double eval(double x, double y) { return x-y; }},
  TIMES { double eval(double x, double y) { return x*y; }},
  DIVIDE { double eval(double x, double y) { return x/y;}};

  abstract double eval(double x, double y);
}
```

Variable number of arguments

...

- „three dots“
- only as the last argument
- either array or list of arguments can be passed
- in the method, available as an array

```
void argtest(Object... args) {
  for (int i=0;i <args.length; i++) {
    System.out.println(args[i]);
  }
}
argtest("Hello", "how", "are", "you");
argtest(new Object[] {"Hello", "how", "are",
  "you"});
```

- methods printf
  - `System.out.printf("%s %d\n", user, total);`

Test

- Are the calls equivalent?

```
argtest("Ahoj", "jak", "se", "vede");
argtest(new Object[] {"Ahoj", "jak", "se", "vede"});
argtest((Object) new Object[] {"Ahoj", "jak", "se",
  "vede"});
```

a) Yes, all of them
b) Only 1. and 2.
c) Only 2. and 3.
d) Each of them will print something different