Konzepte objektorientierter Programmierung

Prof. Dr. Peter Müller

Chair of Programming Methodology

Exercises 7: Aliasing

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Home-work 1

```
class Exa {
  /* invariant \type(a) <: A; */
  Object a;

  /* requires \type(c) <: C; */
  /* ensures \type(\result) <: B; */
  Object m(Object c) { ... }
}
```

Temporary violations of invariant possible!

Home-work 2

```
class Super {
   B m(C c) { ... }
}

class Sub extends Super {
   E m(F c) { ... }
}

C <: F
E <: B
```

Home-work 2

- Assume: F <: C, and B <: E

```
Super s = new Sub();
C c1 = new C();
B b1 = s.m( c1 );
```

Exercise 1

```
class Motor {
   boolean isOK() {/*...*/ }
   boolean isRunning(){/*...*/ }
   void start() { /*...*/ }
   ...
}
class Wheel {
   void deflate() { /*...*/ }
   boolean isOK() {/*...*/ }
   ...
}
class MotorTrouble extends Exception {
   public Motor motor;
   public MotorTrouble( Motor m ) {
     motor = m;
   }
}
```

Exercise 1 – The Invariant

```
public class Car {


  /*@ invariant engine.isRunning() ==>
    @ (\forall int i;
    @      i >= 0 && i < wheels.length;
    @           wheels[i] != null &&
    @           wheels[i].isOK())
    @*/
```

Problem 1 – Public Fields

```
public class Car {
  public Motor engine;
  public Wheel[] wheels;
```

```
Car c = new Car(...);
c.start();
// remove a wheel -> breaks invariant
c.wheels[0] = null;
```

Solution for Problem 1

- Use proper Information Hiding
- Make all fields private or protected

Problem 2 – Capturing

```
public class Car {
  public Car( Motor m, Wheel[] w ) {
     engine = m;
     wheels = w;
  }
```

```
Motor m = new Motor();
Car c = new Car(m, w);
c.start();
// stop the motor, car still thinks it runs
m.stop();
```

Solution for Problem 2 – Capturing

- Never directly store parameters as internal state
- Clone the given objects
- For arrays you need a deep copy, otherwise the array elements might still be aliased

Problem 3 – Leaking through Return Value

```
public Motor getMotor() {
     return engine; }

public Wheel[] getWheels() {
     return wheels; }
```

```
Car c = new Car(m, w);
c.start();
// change wheel while driving
c.getWheels()[0] = new Wheel();
```

Solution for Problem 3 – Leaking

- Never return a reference to the internal state to the outside
- Always clone the objects
- Again for arrays you need a deep copy

Problem 4 – Leaking through Exception

```
public void start() throws MotorTrouble {
  if( engine == null || !engine.isOK() ) {
    throw new MotorTrouble( engine );
  } else { engine.start(); }
}
```

```
try { ... }
catch( MotorTrouble mt ) {
  mt.motor.reset();
}
```

Solution for Problem 4 – Exceptions

- Exceptions can give access to internal state
- Examine what information you really want to share
- Maybe only pass a String with this information

Invariants for Java (Simple Solution)

- Assumption: The invariants of object X may only refer to **private attributes** of X

- For each invariant, we have to show
  - That all exported methods **and constructors of class T** preserve the invariants **of all objects of T and T's subclasses**
  - That all constructors **in addition** establish the invariants of the new object

Invariants can refer to all attributes

```
public class Super {

  private int f;

}
```

```
public class Sub extends Super {

  /*@ invariant f > 0; @*/

}
```

Exercise 2.a: Invariants for Java (Extended Solution)

- Assumption: The invariants of object X may refer to **private and default attributes** of X

- For each invariant, we have to show
  - That all exported methods **and constructors of class T** preserve the invariants **of all objects of all classes in T's package and all subclasses of classes in T's package**
  - That all constructors **in addition** establish the invariants of the new object

Exercise 2.b: Invariants of T objects

```
public class T {
  /* default */ int f;

  /*@ invariant f > 0; @*/

  void violate() { f = -10; }
}
```

Exercise 2.b: Invariants of objects in T's package

```
public class A {
  /* default */ int f;

  /*@ invariant f > 0; @*/
}
```

```
public class T {
  private A myA;

  void violate() { myA.f = -10; }
}
```

Exercise 2.b: Invariants of objects of T's subclasses

```
public class T {
  /* default */ int f;

  void violate() { f = -10; }
}
```

```
public class SubT extends T {

  /*@ invariant f > 0; @*/

}
```

Exercise 2.b: Subclasses of all classes in T's package

```
public class A {
  /* default */ int f; }

public class SubA extends A
  /*@ invariant f > 0; @*/ }
```

```
public class T {
  private A myA;

  void violate() { myA.f = -10; }
}
```

Exercise 3.a

B should provide at least as much access as A

```
class A {
     public invariant f > 0;
     private int f;
}

class B extends A{
    ...
    // invariant f > 0 must holds
    // B can not access f
}
```

Exercise 3.b

A should provide at least as much access as B

```
class A {
     private invariant f > 0;
     public int f;
}

class B extends A{
    ...
    // B can change f
    // invariant f > 0 can not be checked
}
```