Algorithmen, Datenstrukturen und Datenabstraktion
Übung 4

Tutoren: Christopher Filsinger / Fabian Halama

Noah Witte-Winnett Amir Mohammad Gilani

16. November 2018

1. a) encapsulation: this, private inheritance: super, extends

   b) Method Overloading:

```
public int add(int a, int b){
 ‘   return a + b;
}

public double add(double a, double b){
    return a + b;
}
```

Inheritance Polymorphism:

```
class Shape{
    public abstract area;
}

class Circle extends Shape{
...
    double area(){
        return Math.PI * radius * radius
    }
...
}

class Square extends Shape{
...
    double area(){
        return length * length
    }
...
}
```

Generic Data Types

```java
        public class GenDataType<T>{
            T parameter;
            public GenDataType(T parameter){
                this.parameter = parameter;
            }


            public static void main(String[] args){
                GenDataType<Integer> gdt1 = new GenDataType<Integer>(5);
                GenDataType<?> gdt2 = new GenDataType<String>("This is a valid assignme
                GenDataType<? extends Number> gdt3 = new GenDataType<Integer>(69);
                GenDataType<? extends Number> gdt4 = new GenDataType<Float>(453.8883f);
                GenDataType<? super Double> gdt5 = new GenDataType<Number>(13);
            }
        }
```

c) Annotations are pieces of code that can be added to a class, method, variable, parameter or package to specify how these elements should be handled by the compiler and at run-time. Annotations function as meta-data, structuring code so that it is used correctly, providing appropriate warning when the actual use of the code does not conform to the intended one.
The @Override annotation is used when a subclass is implementing a method or variable that is declared in the parent class. @Override demands that the method or variable declared in the subclass overrides a method or variable of the same name declared in the parent class.
(https://en.wikipedia.org/wiki/Java_annotation)

d) Static types are assigned at compile time whereas dynamic types can remain undeclared during compile time but are assigned at run-time. In Java every variable is statically typed but the type can be altered at run-time, e.g. an object of type X can become of a type Y that extends X.

```java
interface I {
    void a();
    }
class A implements I {
    public void a() { System.out.println("A"); }
}
class B implements I {
    public void a() { System.out.println("B"); }
    public void b() { System.out.println("C"); }
}
public static void main(String[] args) {
    A a = new A();  // valid, a of type A implements I. Static type A
    a.a();          // valid, method a() implemented from I : prints "A"
    //a.b();        // invalid, class A has no method b()
    B b = new B();  // valid, b of type B implements I. Static type B
    b.a();          // valid, method a() implemented from I : prints "B"
    b.b();          // valid, method b() from Class B       : prints "C"
    I i;            // valid, i must be initiated as a class that
                    // implements the interface I. Static type I
    //i = new I();  // invalid, I is interface and no new instance of
                    // it can be initialized
    i = a;          // valid, A has the same methods as I.
                    // Static type I, dynamic type A
```

```java
    i.a();              // valid, i of type A has method a, implemented from I
                        // : prints "A"
    //i.b();            // invalid, a has no method b() because class A has
                        // no method b()
    //i = b;            // invalid, b of static type B, which is incompatible
                        // with static type I
    i.a();              // valid, i still of static type I, dynamic type A
                        // prints "A"
    //i.b();            // invalid, i still of dynamic type A
    //b = i;            // invalid, b of static type B, i of dynamic type A
    //b = (B)i;           // invalid, i of static type I, cannot be cast to B
    a = (A)i;           // valid, static and dynamic type A
}
```

What prints is:

```
    A
    B
    C
    A
    A
```

e)

```java
public static void main(String[] args) {
    // valid, List of unknown static type, ArrayList of type String
    List<?> x = new ArrayList<String>();
    // invalid, the type of the elements in the list needs to be
    // declared before the list is intialized, and the declared type
    // must match the initialized type
    //List<Object> y = new ArrayList<Integer>();
    // valid, list of static type ? extends Number (ie Number),
    // dynamic type Integer
    List<? extends Number> y2 = new ArrayList<Integer>();
    // valid, list of Objects, dynamic list of Integers
    Object[] z = new Integer[3];
    // invalid, "ALP3" is of static type String, not Integer
    // z[2] = "ALP3";
}
```

2. a) Show: $\log n \in O(2^{\log\log^2 n})$

$$\begin{aligned} O(2^{\log\log^2 n}) &= O(\log^2 n) \\ \log n &\in O(\log^2 n) \\ \iff \exists n_0 \geq 0,\ c > 1\ \forall n &\geq n_0 : \\ \log n &\leq c \cdot \log^2 n \end{aligned}$$

choose $n_0 = 2$ and $c = 1$
For all $n \geq n_0$ the following is then valid:

$$0 \leq \log n \leq 1 \cdot \log^2 n$$

b) Show: $f(n) \in O(g(n)) \Leftrightarrow g(n) \in \Omega(f(n))$

$$\begin{aligned} f(n) \in O(g(n)) &\Leftrightarrow \exists c, n_0 > 0 : \forall n > n_0 : f(n) \leq c \cdot g(n) \\ &\Leftrightarrow \exists c, n_0 > 0 : \forall n > n_0 : 1c(f(n)) \leq c \cdot g(n) \\ &\Leftrightarrow g(n) \in \Omega(f(n)) \end{aligned}$$

the claim is only valid, when $c^{-1} = \Omega$.

c) Show: $f(n) \in O(g(n)) \Rightarrow 2^{(f(n))} \in O(2^{(g(n))})$
the claim is false.
Counter: assuming $f(n) = 2\log n, g(n) = \log n$
$\Rightarrow O(2\log n) \in O(\log n)$ $f(n) \in O(g(n))$ since they differ only by a constant factor,
yet in the case $2^{f(n)} = 2^{2\log n} = n^2$ the factor is squared,
while in the case of $2^{g(n)} = 2^{\log n} = n$ is not.
$\lightning$

3. a) Every time a bit is flipped from $0 \to 1$ we will pay 2 energy units: 1 unit is used to pay for the current energy expenditure, and 1 unit is handed over to the accountant to save for when the bit has to be flipped back from $1 \to 0$.
Case 1: the right-most bit is a 0
Two power units are paid, one for the flip $0 \to 1$ and one is saved by the accountant. We accrue no debt.
Case 2: the right-most bit is a 1 and there are $k-1$ 1s to the left of it
Previously $k-1$ bits were flipped $0 \to 1$ so we have at least $k-1$ power units

stored. The $k+1$st bit needs to be flipped so we pay two power units, one for the flipping and one to be stored. Altogether we now have at least $k+1$ power units to pay for $k-1$ flips from $1 \to 0$ and one $0 \to 1$ flip, a total of k flips demanding k power units. We accrue no debt and are left with at least $k+1-k=1$ power unit.

b) Table illustrating how much energy is required to count to 8 in binary

Value	2^3	2^2	2^1	2^0
0	0	0	0	0
				+1
1	0	0	0	1
			+1	+1
2	0	0	1	0
				+1
3	0	0	1	1
		+1	+1	+1
4	0	1	0	0
				+1
5	0	1	0	1
			+1	+1
6	0	1	1	0
				+1
7	0	1	1	1
	+1	+1	+1	+1
8	1	0	0	0
Total Energy	$1=2^0$	$2=2^1$	$4=2^2$	$8=2^3$

.

From the table we can see that the worst case for counting in binary is when you are incrementing a number 2^i-1, which is represented in binary as a string of k 1s. Adding 1 will flip each $1 \to 0$ in addition to the $k+1$st bit being flipped $0 \to 1$. In our example, the total energy expended when counting from 0 to $n=8=2^3$ in binary, so from 0000 to 1000 is

$$8+4+2+1=2^3+2^2+2^1+2^0=\sum_{i=0}^{3} 2^i = \sum_{i=0}^{log_2(n)} 2^i = 15$$

Assuming that we are counting to a number that is to the power of 2 we can therefore assume $n=2^l$ and the total power expenditure is

$$\sum_{i=0}^{l} 2^i = 2^{l+1}-1 = 2 \cdot 2^l - 1 = 2 \cdot n - 1$$

Therefore $T(n)=2n-1$, which means $T(n) \in O(n)$