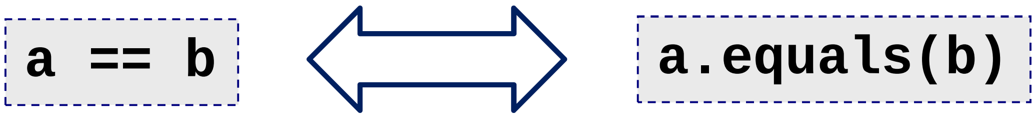
Basics of programming 3

Java utilities

Objects’ equality

- `==` operator
  - reference based equality
- **`boolean equals(Object o)`**
  - content based equality
  - recursion advised
  - *default implementation is reference based*



`a == b`

`a.equals(b)`

Comparisons

- Natural
  - □ implements interface `Comparable<T>`
  - □ `int compareTo(T o)`

    this Δ o ↔ this.cmp(o) Δ 0
  - □ single implementation per class
  - □ set at compile time
    - tricks are allowed ☺

Comparisons

- Comparator based
  - *Strategy* pattern: responsibility separately
  - `interface Comparator<T>`
  - `int compare(T o1, T o2)`
    - compares *T*-s

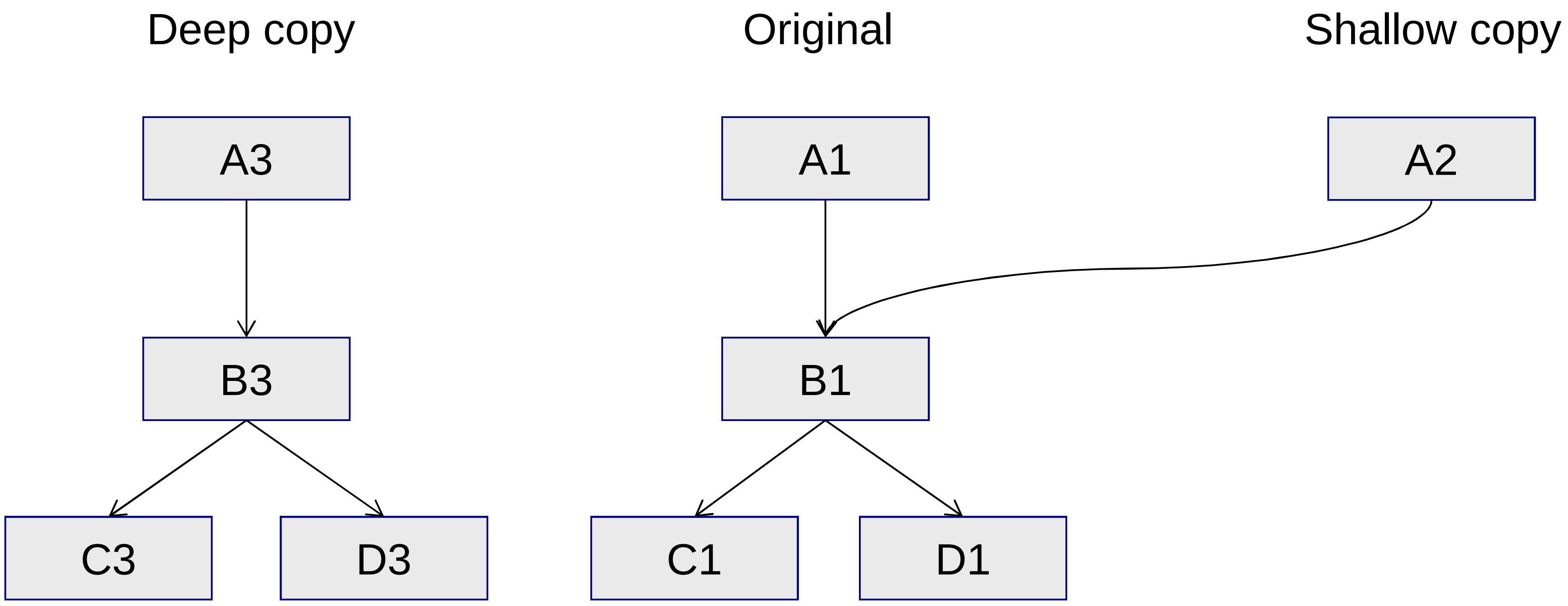
$$o1 \ \Delta \ o2 \ \leftrightarrow \ cmp(o1,o2) \ \Delta \ 0$$

  - `boolean equals(Object obj)`
    - compares *Comparators*

Copying objects

- Implementing **`java.util.Cloneable`**
- Overriding **`Object.clone()`**
  - calling **`super.clone()`** is advised
  - *Object.clone()* is tricky: instantiates subclass
- *Shallow copy*
  - Only references are copied
    - e.g. Copy of a Vector has references to the original objects
- *Deep copy*
  - recursive copy
    - e.g. correct String implementation in C++

Deep and shallow copy



Copy: no superclass, naïve

```
public class A { // this example is a
                 // naïve implementation
      B b;
      public Object clone() { // shallow
            A a2 = new A();
            a2.b = b;
            return a2;
      }
      public Object clone() { // deep
            A a3 = new A();
            a3.b = b.clone();
            return a3;
      }
      ...
}
```

Copy: superclass declared

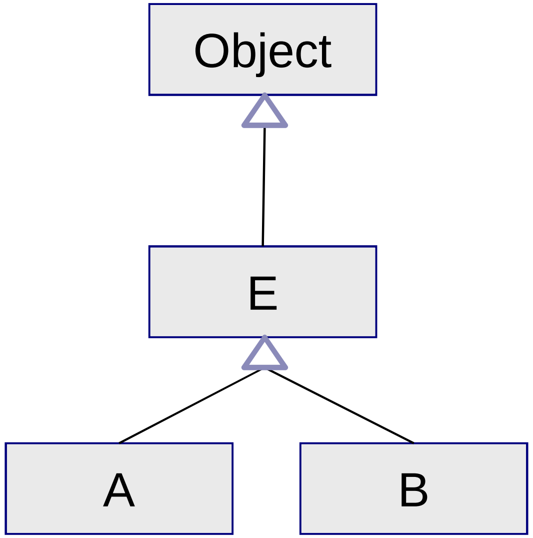
```
public class A extends E {
      B b;
      public Object clone() { // shallow
            A a2 = (A)super.clone(); // !!!
            a2.b = b;
            return a2;
      }

      public Object clone() { // deep
            A a3 = (A)super.clone();
            a3.b = b.clone();
            return a3;
      }
      ...
}
```

Calls *clone()* in superclass, creates object of *class A*

Clone implementation

```
public class E {
  public Object clone() {
    return super.clone(); // needed!
  }
}
public class A extends E {
  B b;
  public Object clone() {
    A a3 = (A)super.clone();
    a3.b = b.clone(); // deep
    return a3;
  }
  ...
}
public class B extends E { ... }
```



Copy constructor?

```
public class A {
      B b;
      public A(A a) {
            this = a.clone(); // DON’T!!!
      }
      public A(A a) {
            this.b = a.b.clone(); // ctr vs clone
      }
      public A(A a) {
            this.b = new B(a.b); // inheritance?
      }
      ...
}
```

Deep clone vs. Copy ctr

- (Java vs C++)

	Pros	Cons
Deep clone	works with abstract classes	**new** is omitted -> who is allocating memory (C++)?
Copy ctr	homogeneous, uses **new;** **delete** is OK (C++)	problem with abstract classes

Fast identity: *hash*

- *public int hashCode()*
  - returns object-specific int
    - *a.equals(b)==true* → *a.hashCode() == b.hashCode()*
    - purpose: store and find objects effectively
      - e.g. HashMap, HashSet
  - possible implementation in *Object*: memory address
- Good hash function is an art
  - minimize clustering, etc
  - more details in *Theory of Algorithms*

Implementing hash function

```
class Test {
       Object o1; // any kind of object
       Object o2;
       ...
       Object on;
       public int hashCode() {
              int h = 0;

              h = 31*h+o1.hashCode(); // recursion
              h = 31*h+o2.hashCode();
              ...
              h = 31*h+on.hashCode();

              return h;
       }
}
```

Enum: an object type

- Enums have their own class
    - attributes
    - methods
- Enums are
    - serializable
    - printable
    - for-each-able
    - switch-case-able

```
public enum Planet {
    Mercury, Venus, Earth, Mars,
    Jupiter, Saturn, Uranus, Neptune
}
```

Enum example

```
public enum Planet { // complex enum
    Mercury(3.3e23, 2.44e6), Venus(4.868e24, 6.052e6),
      Earth(5.972e24, 6.371e6), Mars(6.417e23, 3.39e6),
      Jupiter(1.899e27, 6.991e7), Saturn(5.684e26, 5.823e7),
      Uranus(8.681e25, 2.536e7), Neptune(1.024e26, 2.462e7);

    private final double mass, radius; // kg, m

    Planet(double m, double r) { mass = m; radius = r;}

    public double mass() { return mass; }
    public double radius() { return radius; }
    public double sGrav() { return 6.674e-11*mass/radius/radius; }
}
```

```
for (Planet p : Planet.values()) {
        System.out.println(p+": "+p.sGrav());
}
```

Enum example 2

```
enum Letter {A, B, C}
```

```
Letter e = Letter.A;
switch (e) {
      case A: System.out.println("A!"); break;
      case B: System.out.println("B!"); break;
      case C: System.out.println("C!"); break;
}
```

Static import of members

```
import static mypackage.Letter.*;
```

```
if (e == B) { ... }
```

Enum methods

- `String name()`
  - name of the enum constant
- `int ordinal()`
  - serial number
- `static <T extends Enum<T>> T valueOf([Class<T> enumType,] String name)`
  - returns enum const from (type and) name
- `static <T extends Enum<T>> T[] values()`
  - returns the constants of the enum type

Variable parameters

- Pre 1.5: array is passed
  - uncomfortable

```
void foo(String s, Object[] oa) { for (Object o : oa) …; }
```

- 1.5: *varargs*
  - both arrays and sequences are accepted
  - at the end of parameter list only

```
void foo(String s, Object… oa) { for (Object o : oa) …; }
```

```
Object[] oo = {"a", "b", "c"};
foo("X", oo);
foo("X", "j", "k", "l", "m", "n");
```

Annotations

- Adding plus information to the source code

```
public @interface Copyright {
    String value() default "2008 Me";
}
@Copyright("2008 Bytemongers Limited")
public class ÁrvíztűrőTükörfúrógép { ... }
```

  - *Declaration: interface starting with a @*
  - *Methods describe the annotation's members*
    - *return value primitive, String, Class, enum*
  - *Members might have a default value*

Using annotations

- Annotations describe metadata
  - □ used by compilers
  - □ code generators
  - □ etc.
- E.g.: method overload:

```
@Override
public int read() throws IOException {
      return super.read();
}
```

Utility classes

- String handling
  - *StringTokenizer*: splits strings into tokens
  - *StringBuffer*, *StringBuilder*: *effective string handling*
- Calendar handling
  - *Date*, *Calendar*, *GregorianCalendar*
- Mathematics
  - *Random*: generating random numbers
  - *Math*, *StrictMath*: math functions (sin, exp, etc)
- Scanner
  - helps reading from streams

StringBuffer and StringBuilder

- Mutable string representations
- Used for string-intensive operations
  - concatenation (append, concat, insert, etc.)
  - character modification, deletion, etc.
- StringBuffer
  - multithreaded, slower
- StringBuilder
  - single-threaded, faster

StringTokenizer

- Problem
  - Parsing lines
    - configuration files
    - scripts
    - etc.
- Solution
  - Splitting lines into tokens (words)
    - *String.split*: String to array
    - *StringTokenizer*: String to tokens
  - Delimiter to be specified

StringTokenizer

- Constructors
  - tokenizer has to be initialized first
  - StringTokenizer(String str)
    - delimiters: space, tab, newline, carriage-return, form-feed
  - StringTokenizer(String str, String delim)
    - sets delimiter characters as well
  - StringTokenizer(String s, String d, boolean retDels)
    - sets delimiters and returns them

StringTokenizer

- int count tokens()
  - returns number of tokens
- boolean hasMoreElements
- boolean hasMoreTokens
  - returns if tokenizer has more tokens
- Object nextElement
- String nextToken
  - next token is returned
- String nextToken(String delim)
  - next token with changed delimiters (prevails)

StringTokenizer example

```
StringTokenizer st =
      new StringTokenizer("alpha beta gamma");
while (st.hasMoreTokens()) {
      System.out.println(st.nextToken());
}
```

```
alpha
beta
gamma
```

Time-related classes

- Date
  - represents an instant in time
    - millisec precision
  - mostly deprecated
- Calendar
  - abstract
  - for conversion between dates and time fields
- GregorianCalendar *extends Calendar*
- DateFormat
  - for formatting and parsing date strings

Date

- Date() and Date(long d)
  - ctr-s for *now* and *d* (ms since epoch 1970-01-01UTC00:00:00)
- boolean after/before(Date d)
- int compareTo(Date d)
- int equals(Object o)
  - trivial comparisons
- long getTime()
  - ms since epoch
- String toString()
  - returns time in “dow mon dd hh:mm:ss zzz yyyy” format

Calendar

- Represents a date
  - abstract class
  - static method *getInstance()* returns current date
- Handling with generic methods
  - *add(f, delta)*
  - *set(f, delta), get(f), clear(f)*
  - *roll(f, delta)*
    - adds, sets, rolls, gets, clears field *f* (with *delta)*
    - adjusts if necessary
    - *f* is specified by constant values

Calendar

- Constants for fields
  - DATE, DAY_OF_MONTH, DAY_OF_WEEK, DAY_OF_YEAR
  - WEEK_OF_MONTH, WEEK_OF_YEAR
  - ERA, YEAR, MONTH, MINUTE, SECOND, MILLISECOND
  - AM_PM, HOUR_OF_DAY (0-24), HOUR (0-12)
  - ZONE_OFFSET
- Constants for values
  - JANUARY, FEBRUARY, MARCH, etc
    - starts with JANUARY==0 **!!!**
  - MONDAY, TUESDAY, etc
    - SUNDAY==0, MONDAY==1, etc
  - AM, PM
  - (GregorianCalendar: AD, BC)

Calendar

- Methods for everything
  - boolean after/before(Object when)
  - int clear()
  - int get[Actual]Maximum(f)
  - int get[Actual]Minimum(f)
  - int getGreatestMinimum(f)
  - int getLeastMaximum(f)
  - get/setFirstDayOfWeek
  - get/setMinimalDaysInFirstWeek
  - …

GregorianCalendar

- Extends Calendar
  - Constructors
    - year, month, day *[,hour, minute [, second]]*
  - Constants
    - *AD, BC*
  - Methods
    - setGregorianChange(Date date)
    - Date getGregorianChange()
    - boolean isLeapYear()
    - …

Calendar example

```
Calendar c = new GregorianCalendar(1996,0,23);//96-01-23
c.set(Calendar.MONTH, Calendar.MAY); // 1996-05-23
c.set(Calendar.DATE, 31);         // 1996-05-31

c.add(Calendar.MONTH, 15);        // 1997-08-31
c.roll(Calendar.DATE, 10);        // 1997-08-10

DateFormat df = DateFormat.getDateTimeInstance();
System.out.println(df.format(c.getTime()));
   // Aug 10, 1997 12:00:00 AM
```

Random

- For generating random numbers
  - constructors
    - *default*: seed automatically set
    - *seeded* (param *long*), deterministic *(setSeed)*
  - *nextXXX()*
    - uniform distribution
    - *boolean*, *bytes*, *int*, *long*: result in type's range
    - *double*, *float*: result in range [0.0, 1.0)
    - *nextInt(int n)*: between 0 and *n* (exclusive)
  - *nextGaussian()*
    - normal distribution (mean: 0, std dev: 1)

Math/StrictMath

- Utility classes
  - java.lang.Math, java.lang.StrictMath
- Contants
  - E, PI
- Functions
  - abs, signum, cbrt, sqrt, ceil, floor, round, rint,
  - sin, cos, tan, sinh, cosh, tanh
  - asin, acos, atan, atan2
  - pow, exp, expm1, log, log10, log1p, scalb,
  - max, min, nextAfter, nextUp, toDegrees, toRadians

Big numbers

- BigDecimal
- BigInteger

Scanner

- Character based input
  - BufferedReader
    - awkward
    - complex
    - nonlegible
  - Scanner
    - direct access to resource
    - iterator-like usage
    - conversion to primitive types

Scanner

- Constructors
  - params: *File*, *InputStream*, *Path*, *Readable*, *String*
- Methods
  - *hasNext[XXX]*
  - *next[XXX]*
    - BigDecimal, BigInteger, boolean, byte, double, …
  - *useDelimiter(Pattern|String), delimiter()*
  - *useRadix(int radix), radix()*
    - for setting/getting delimiter and radix
  - *findInLine, skip, match, etc*