

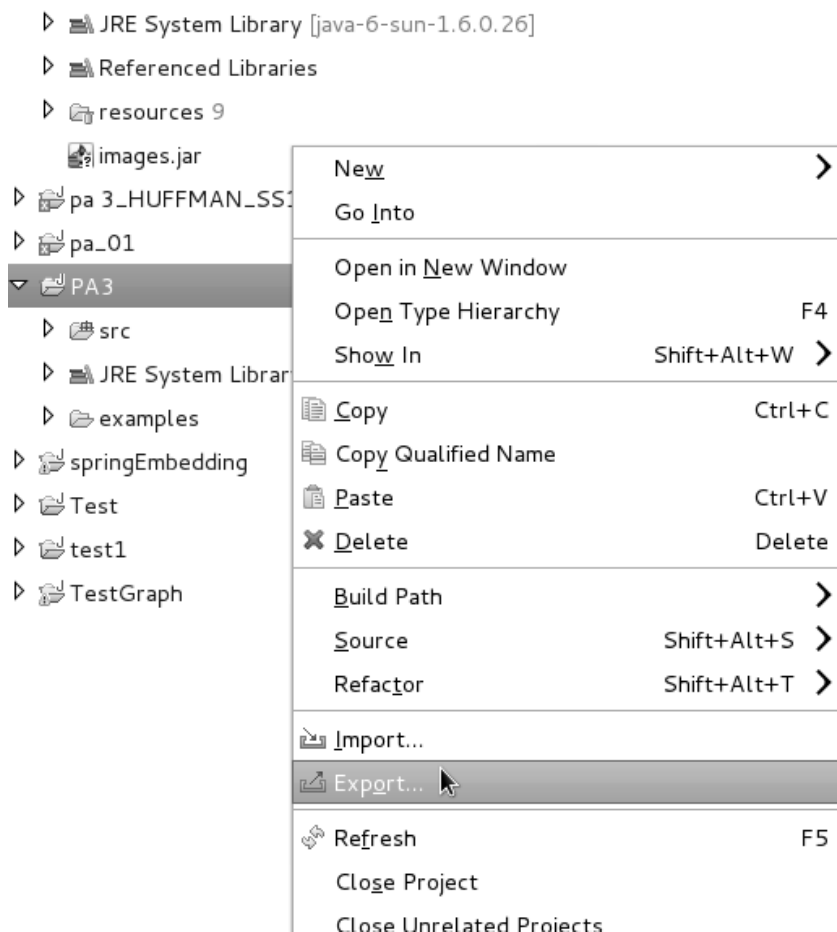
7. Übung

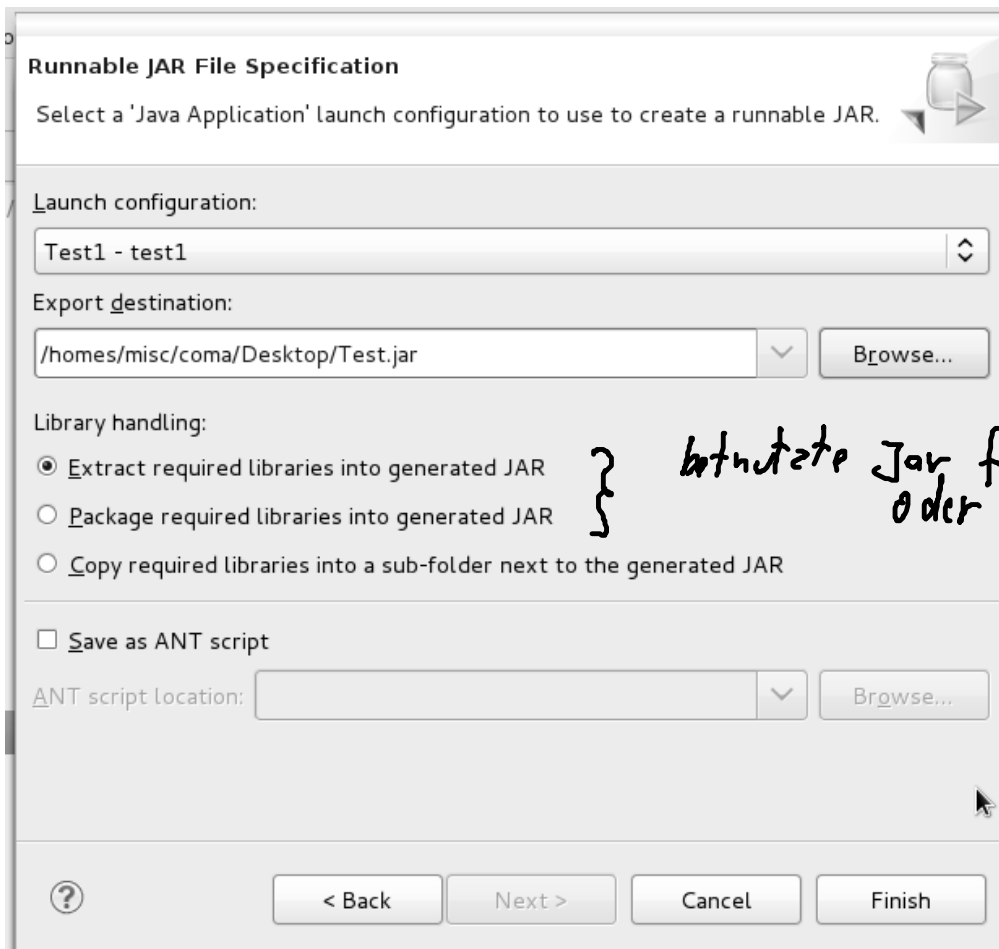
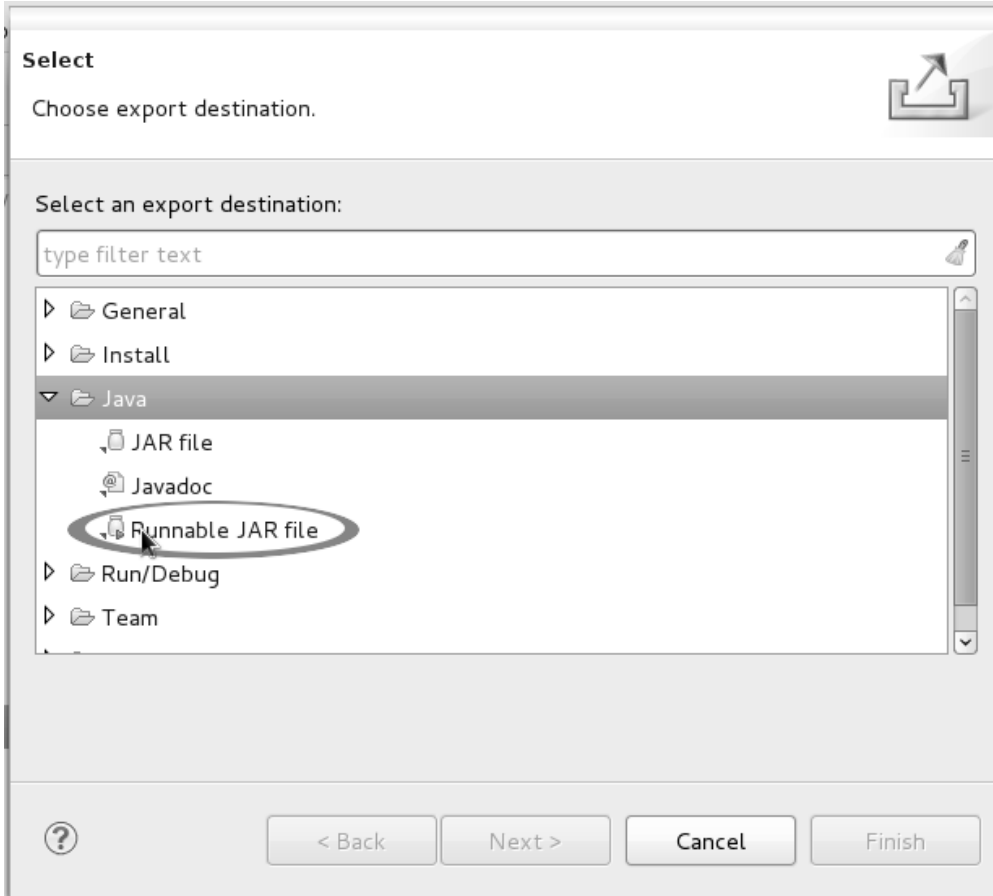
## Jars, Hashing in Java

**Jar:** Java-Archive, ZIP Datei mit Java byte code  
(ähnlich exe-Datei)

runnable: zusätzliche Informationen zum Ausführen  
META-INF/MANIFEST.MF: welche Main(),  
Classpath inform.

Erzeugen: per Konsole „jar“ + eigenes Manifest  
einfacher: per Eclipse:





} Auswahl der Main Methode

} benutzte jar files entpacken oder nur hinzufügen

Projektabgabe: Jar Archiv mit folgender Syntax:

java -jar Fir na 135 KI.jar <Server addr.> <Port>  
<First name> <last name> <Passwort> <Spielname>  
<Spielfarbe>

KI: zum Server mit <Serveraddr.> verbinden unter

- Port <Port>
- die Namen <First name> <last name> auf Server nutzen
- bei Spiel farbe weiss: Spiel <Spielname> mit Passwort auf Server erstellen

Schwarz: zu Spiel <Spielname> mit <passwort> verbinden

## Hashing:

- Abbildung von großer Schlüsselmenge auf kleine Zielmenge (Hash tabellen position)
- Speichere Schlüssel in Array, effiziente Positions-<sup>bestimmung</sup>
- Strategien bei Kollisionen: Chaining, Open addressing

Java: Methode hashCode(): Object  $\rightarrow$  int (eigene Klassen: überschreiben, sonst nicht sinnvoll)

Regeln: • equals() implementiert  $\rightarrow$  auch hashCode() überschreiben

• equals und hashCode dieselben Felder berücksichtigen

•  $a \text{ equals } b \Rightarrow a.\text{hashCode}() == b.\text{hashCode}()$

~~X~~

Beispiel: hashCode() Implementation der Klasse String

- bis Java 1.2, max. 16 bits berücksichtigt um schnell auswertbar zu sein

API:

## hashCode

```
public int hashCode()
```

Returns a hash code for this string. The hash code for a String object is computed as

$$s[0]*31^{(n-1)} + s[1]*31^{(n-2)} + \dots + s[n-1]$$

using int arithmetic, where  $s[i]$  is the  $i$ th character of the string,  $n$  is the length of the string, and  $\wedge$  indicates exponentiation. (The hash value of

**Overrides:**

`hashCode` in class `Object`

**Returns:**

a hash code value for this object.

**See Also:**

`Object.equals(java.lang.Object)`, `Hashtable`

Java: String  $s$

$$h(s) = \sum_{i=0}^{n-1} s[i] \cdot 31^{(n-i-1)}$$

Interpretation des Strings  
als Zahl in Basis 31,  
aber  $s[i] \in \{0, 255\}$

$n$  = Länge des  
Strings  
 $s[i]$  =  $i$ -te Stelle  
des Strings

Beispiel:

- "\_"  $\rightarrow$  32
- "a"  $\rightarrow$  97
- "b"  $\rightarrow$  98
- "CoMoCoMo"  $\rightarrow$  -1465 99744 Kollision
- "991939195"  $\rightarrow$

Frage: Wie entwerfe ich `hashCode()` für eigene Klasse?

wichtig: Bitoperationen:

$\wedge$  = xor, exklusives oder  $1011 \wedge 1101 = 0110$

$|$  = or, oder  $1011 | 1101 = 1111$

$\&$  = and, und  $1011 \& 1101 = 1001$

$\gg$  = bitshift (vorzeichenbehaftet)  $1011 \gg 2 = \boxed{0010}11$

$\ggg$  = bitshift

bitweise Darstellung

Vorschlag für `hashCode()`:

```

/**
 * booleans.
 */
public static int hash( int aSeed, boolean aBoolean ) {
    System.out.println("boolean...");
    return firstTerm( aSeed ) + ( aBoolean ? 1 : 0 );
}

/**
 * chars.
 */
public static int hash( int aSeed, char aChar ) {
    System.out.println("char...");
    return firstTerm( aSeed ) + (int)aChar;
}

/**
 * ints.
 */
public static int hash( int aSeed , int aInt ) {
    /*
     * Implementation Note
     * Note that byte and short are handled by this method, through
     * implicit conversion.
     */
    System.out.println("int...");
    return firstTerm( aSeed ) + aInt;
}

/**
 * longs.
 */
public static int hash( int aSeed , long aLong ) {
    System.out.println("long...");
    return firstTerm(aSeed) + (int)( aLong ^ (aLong >>> 32) );
}

/**
 * floats.
 */
public static int hash( int aSeed , float aFloat ) {
    return hash( aSeed, Float.floatToIntBits(aFloat) );
}

/**
 * doubles.
 */
public static int hash( int aSeed , double aDouble ) {
    return hash( aSeed, Double.doubleToLongBits(aDouble) );
}

```

Funktion für alle prim. Datentypen

Idee: zu Anfangshash ein weiteres Feld hinzufügen  
→ neuen hash



erschicht um die Hälfte  
→ verknüpft erste und zweite Hälfte des longs

als bits interpretieren

```

public static int hash( int aSeed , Object aObject ) {
    int result = aSeed;
    if ( aObject == null ) {
        result = hash(result, 0);
    }
    else if ( ! isArray(aObject) ) {
        result = hash(result, aObject.hashCode());
    }
    else {
        int length = Array.getLength(aObject);
        for ( int idx = 0; idx < length; ++idx ) {
            Object item = Array.get(aObject, idx);
            //recursive call!
            result = hash(result, item);
        }
    }
    return result;
}

```

allgemeines Object

```

/// PRIVATE ///
private static final int FODD_PRIME_NUMBER = 37;

private static int firstTerm( int aSeed ){
    return FODD_PRIME_NUMBER * aSeed;
}

private static boolean isArray(Object aObject){
    return aObject.getClass().isArray();
}

```

bsp: 'true' → 1

$$\text{hash}(1, 4) = 37 \cdot 1 + 4$$

seed ↑      ↑ nächstes Feld

# Anwendung in eigener Klasse:

```
/**
 * The following fields are chosen to exercise most of the different
 * cases.
 */
private boolean fIsDecrepit;
private char fRating;
private int fNumApartments;
private long fNumTenants;
private double fPowerUsage;
private float fWaterUsage;
private byte fNumFloors;
private String fName; //possibly null, say
private List fOptions; //never null
private Date[] fMaintenanceChecks; //never null
private int hashCode;
```

Variablen der Klasse  
→ speichert Hashwert nach Berechnung

```
@Override public int hashCode() {
 //this style of lazy initialization is
 //suitable only if the object is immutable
 if ( hashCode == 0 ) {
  int result = hashCodeUtil.SEED;
  result = hashCodeUtil.hash( result, fIsDecrepit );
  result = hashCodeUtil.hash( result, fRating );
  result = hashCodeUtil.hash( result, fNumApartments );
  result = hashCodeUtil.hash( result, fNumTenants );
  result = hashCodeUtil.hash( result, fPowerUsage );
  result = hashCodeUtil.hash( result, fWaterUsage );
  result = hashCodeUtil.hash( result, fNumFloors );
  result = hashCodeUtil.hash( result, fName );
  result = hashCodeUtil.hash( result, fOptions );
  result = hashCodeUtil.hash( result, fMaintenanceChecks );
  hashCode = result;
 }
 return hashCode;
}
```

verarbeiten aller Felder in Hashcode

## HashSet<E> als Klasse für Hashing Daten:

Constructor Summary	
<b>HashSet()</b>	Constructs a new, empty set, the backing HashMap instance has default initial capacity (16) and load factor (0.75).
<b>HashSet(Collection&lt;? extends E&gt; c)</b>	Constructs a new set containing the elements in the specified collection.
<b>HashSet(int initialCapacity)</b>	Constructs a new, empty set, the backing HashMap instance has the specified initial capacity and default load factor (0.75).
<b>HashSet(int initialCapacity, float loadFactor)</b>	Constructs a new, empty set, the backing HashMap instance has the specified initial capacity and the specified load factor.

Method Summary	
boolean	<b>add(E e)</b> Adds the specified element to this set if it is not already present.
void	<b>clear()</b> Removes all of the elements from this set.
Object	<b>clone()</b> Returns a shallow copy of this HashSet instance: the elements themselves are not cloned.
boolean	<b>contains(Object o)</b> Returns true if this set contains the specified element.
boolean	<b>isEmpty()</b> Returns true if this set contains no elements.
Iterator<E>	<b>iterator()</b> Returns an iterator over the elements in this set.

boolean	<u>remove(Object o)</u> Removes the specified element from this set if it is present.
int	<u>size()</u> Returns the number of elements in this set (its cardinality).

Hash Set kann contains, add, remove und iteration  
- kein get() (wäre auch nicht sinnvoll)

HashMap <K, V>; Key, Value Paar,  
suchen nach Key und Wert erhalten möglich  
hashCode()

Was macht Java intern:

- Kollisionen: chaining
- Speicherung Schlüssel & Wertepaare ab (Schlüssel zum Suchen & beim Vergrößern notwendig)
- Suchen:
  - hashCode() von Key ausrechnen
  - Hashwert auf Tabellenplatz hashen
  - Liste in Tabellenplatz durchlaufen und Schlüssel vergleichen
  - verdoppelt interne Tabelle sobald Füllstand erreicht (75% der Tabellenlänge), danach Schlüssel-Wertepaare neu verteilen
  - interne Zuordnung von Hashwert zu Tabellenplatz:

SourceCode aus java.util.HashMap:

```

/**
 * Applies a supplemental hash function to a given hashCode, which
 * defends against poor quality hash functions. This is critical
 * because HashMap uses power-of-two length hash tables, that
 * otherwise encounter collisions for hashCodes that do not differ
 * in lower bits. Note: Null keys always map to hash 0, thus index 0.
 */
static int hash(int h) {
    // This function ensures that hashCodes that differ only by
    // constant multiples at each bit position have a bounded
    // number of collisions (approximately 8 at default load factor).
    h ^= (h >>> 20) ^ (h >>> 12);
    return h ^ (h >>> 7) ^ (h >>> 4);
}

/**
 * Returns index for hash code h.
 */
static int indexFor(int h, int length) {
    return h & (length-1);
}

```

$length = 2^k, \Rightarrow length-1 = 2^k - 1$   
 $= \underbrace{11111111}_{k \text{ bits}}$   
 $= \boxed{h \% (length-1)}$   
 Index für Key k:  $indexFor(hash(k.hashCode(), length))$

# Hashing hat 3 große Anwendungen

- Datenbanken:
  - Schnellen Zugriff auf Daten
  - mehrere Hashfunktionen möglich pro Datensatz
- Prüfsummen: (Testen von Übertragungsfehlern)
  - wichtig:
    - schnell berechenbar
    - sensibel auf kleine Fehler
    - Beispiel: CRC 32 (cyclic redundancy check)  
zyklische Redundanzprüfung

CRC 32: Idee: Bits als Polynom auffassen und per Polynomdivision den Rest bestimmen

$$\text{bits: } 101101 \rightsquigarrow x^5 + x^3 + x^2 + 1$$

Beispiel: Nachricht: 101101  
Generator polyn: 1101

$$\begin{array}{r} 101101 \\ - 1101 \\ \hline 1100 \\ - 1101 \\ \hline 1100 \\ 1101 \\ \hline 01 \end{array}$$

Rest der Division: 01

Nachricht + Rest = 10110101  
zu übertragen

Test beim Empfänger:

$$\begin{array}{r} 10110101 \\ 1101 \\ \hline 1100 \\ 1101 \\ \hline 1101 \\ 1101 \\ \hline 0000 \end{array}$$

→ Prüfsumme korrekt

falsche Nachricht:



$$\begin{array}{r}
 1 \textcircled{1} 1 1 0 1 0 1 \\
 \underline{1 1 0 1} \\
 1 1 0 1 \\
 \underline{1 1 0 1} \\
 0 1
 \end{array}$$

→ Prüfsumme falsch

- CRC32: - festgelegtes Generator polyn. mit 32 bits
  - verwendet in Zipprogrammen
  - mittlerweile Hardware implementationen
  - Java API hat CRC32 berechnen
- weitere Anwendung: Kryptographie (Signaturen, Passworthasches)