Java Strings

Lecture 15

Immutable String

- Objects of the ***String*** class are **immutable**
- Every method in the class that appears to modify a ***String*** actually creates and returns a brand new ***String*** object containing the modification. The original ***String*** is left untouched.
- Because a **String** is **read-only**, there's no possibility that one reference will change something that will affect the other references.

Java String API (subset)

Method	Parameter	Return value
length( )		Number of characters in the **String.**
charAt( )	**int** Index	The **char** at a location in the **String.**
toCharArray( )		Produces a **char[]** containing the characters in the **String.**
equals( ), equalsIgnoreCase( )	A **String** to compare with	An equality check on the contents of the two **Strings.**
contains( )	A **CharSequence** to search for	Result is **true** if the argument is contained in the **String.**
substring( ) (also **subSequence( )**)	Overloaded: starting index; starting index + ending index.	Returns a new **String** object containing the specified character set.
replace()	The old character to search for, the new character to replace it with. Can also replace a **CharSequence** with a **CharSequence**.	Returns a new **String** object with the replacements made. Uses the old **String** if no match is found

The methods work on the original String but do not change it

```
public class Immutable {
      public static String upcase(String s) {
            return s.toUpperCase();
      }

      public static void main(String[] args) {
            String q = "howdy";
            print(q); // howdy
            String qq = upcase(q);
            print(qq); // HOWDY
            print(q); // howdy
      }
}
```

Passing by value (passing by copy)

When you call Java method with reference variable as a parameter, **a copy of the reference is created** – it means that there are now two different reference variables which point to the same object.

If you assign copy to a different object, this does not change the original object

Passing object references

Thus, we cannot make the object parameter refer to a different object by reassigning the reference or calling *new* on the reference. For example the following method would not work as expected:

```
public static void changeTuple(Tuple t)
{
    t=new Tuple(1,"changed");
}
```

Passing String as a parameter: the same

When you pass a String as a parameter, it is the reference to the original String which is copied into the parameter, but you cannot change any field of the original String because it is immutable – **does not have methods to change its fields**.

What is printed

```
public class ImmutableObjects {
      public static void changeTuple(Tuple t) { t=new Tuple(1,"changed"); }

      public static void changeTupleFields(Tuple t){
        t.setI(1);
        t.setS("changed");
      }

      public static void changeInteger(Integer n) { n=new Integer(1); }

      public static void changeString(String s) { s="changed"; }

      public static void main(String [] args){
        String myS="original";
        changeString(myS);
        System.out.println(myS);

        Integer myN=0;
        changeInteger(myN);
        System.out.println(myN);

        Tuple myT= new Tuple(0,"original");
        changeTuple(myT);
        System.out.println(myT);

        changeTupleFields(myT);
        System.out.println(myT);
      }
}
```

Concatenating strings

- The operator '**+**' has been **overloaded** for ***String*** objects.
- *Overloading* means that an operation has been given an extra meaning when used with a particular class.

(The '+' and '+=' for **String** are the only operators that are overloaded in Java, and Java does not allow the programmer to overload any others.)

Each time creates a new String

```
String mango = "mango";
String s = "abc" + mango + "def" + 47;
```

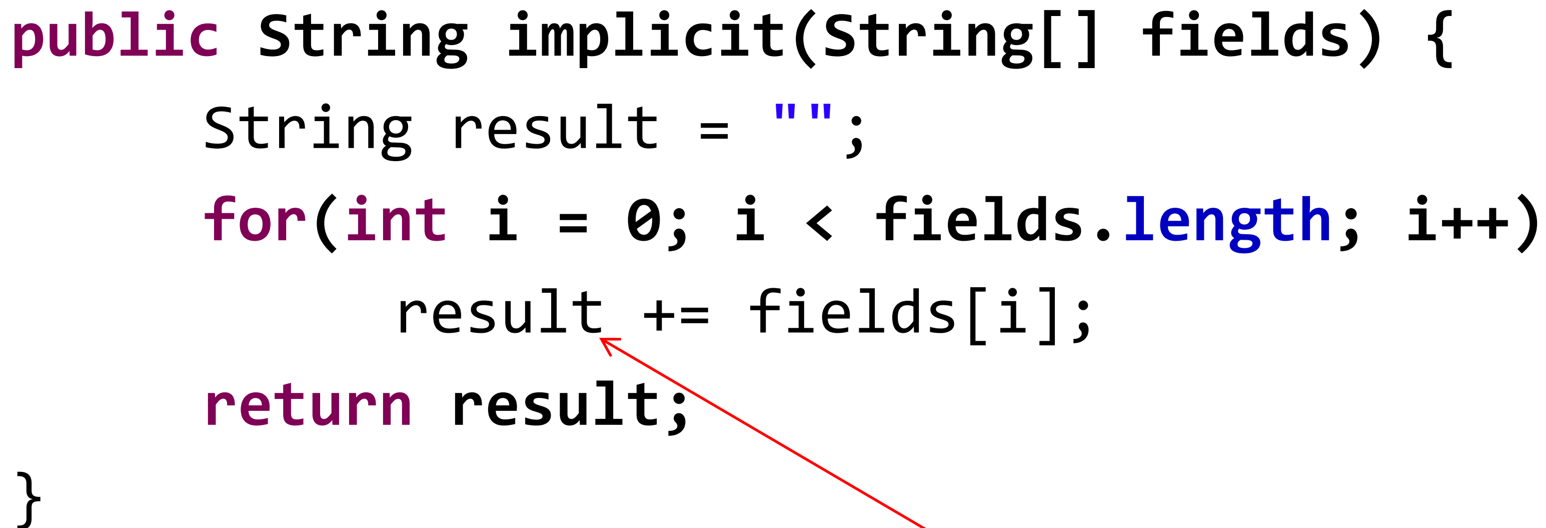
- Creates new strings
  - s0=”abc”
  - s1=“abcmango”
  - s2=“abcmangodef”
  - s=“abcmangodef47”

Need to be garbage collected

This would give unacceptable performance

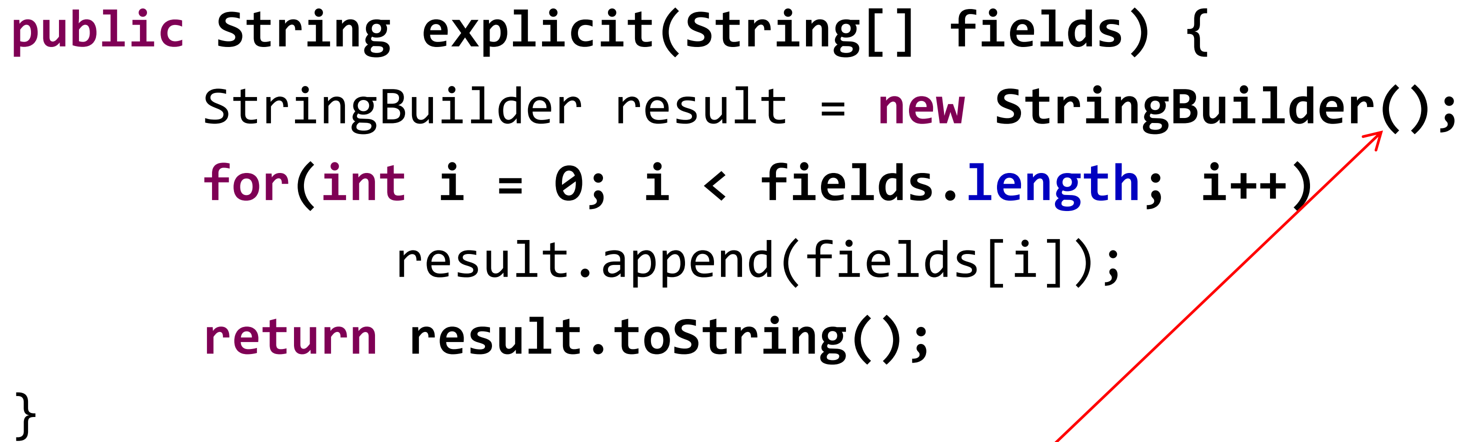
JVM implicitly uses ***StringBuilder***.
An ***append*** method of the ***StringBuilder*** class is called 4 times instead.

StringBuilder is used implicitly



The implicit ***StringBuilder*** construction happens *inside* this loop, which means you're going to get a new ***StringBuilder*** object every time you pass through the loop.

Use StringBuilder explicitly



```
public String explicit(String[] fields) {
        StringBuilder result = new StringBuilder();
        for(int i = 0; i < fields.length; i++)
                result.append(fields[i]);
        return result.toString();
}
```

The method only creates a single **StringBuilder** object.

Creating an explicit **StringBuilder** also allows you to pre-allocate its size if you have extra information about how big it might need to be, so that it doesn’t need to constantly reallocate the buffer.

`toString` method

When you create a **`toString( )`** method:

- If the operations are simple ones that the compiler can figure out on its own, you can generally rely on the compiler to build the result.
- If **looping** is involved, you should **explicitly** use a **`StringBuilder`** in your **`toString( )`**

Example: using StringBuilder in toString()

```
public class RandomSequence25
{
        public static Random rand = new Random(47);

        public String toString() {
                StringBuilder result = new StringBuilder("[");
                for(int i = 0; i < 25; i++) {
                        result.append(rand.nextInt(100));
                        result.append(", ");
        }
        result.delete(result.length()-2, result.length());
        result.append("]");
        return result.toString();
}
```

Creating HashMap of Strings

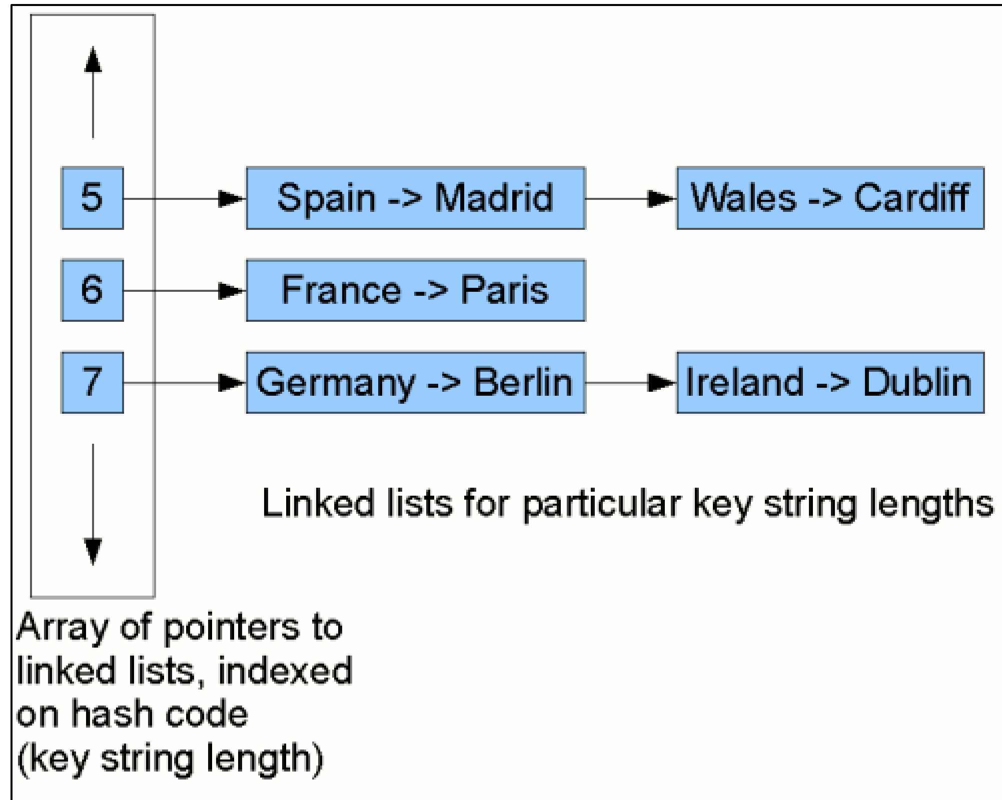
Key	Value
Cuba	Havana
England	London
France	Paris
Spain	Madrid
Switzerland	Berne

Hashing means using some function or algorithm to map object data to some representative integer value.

Hashing by String length

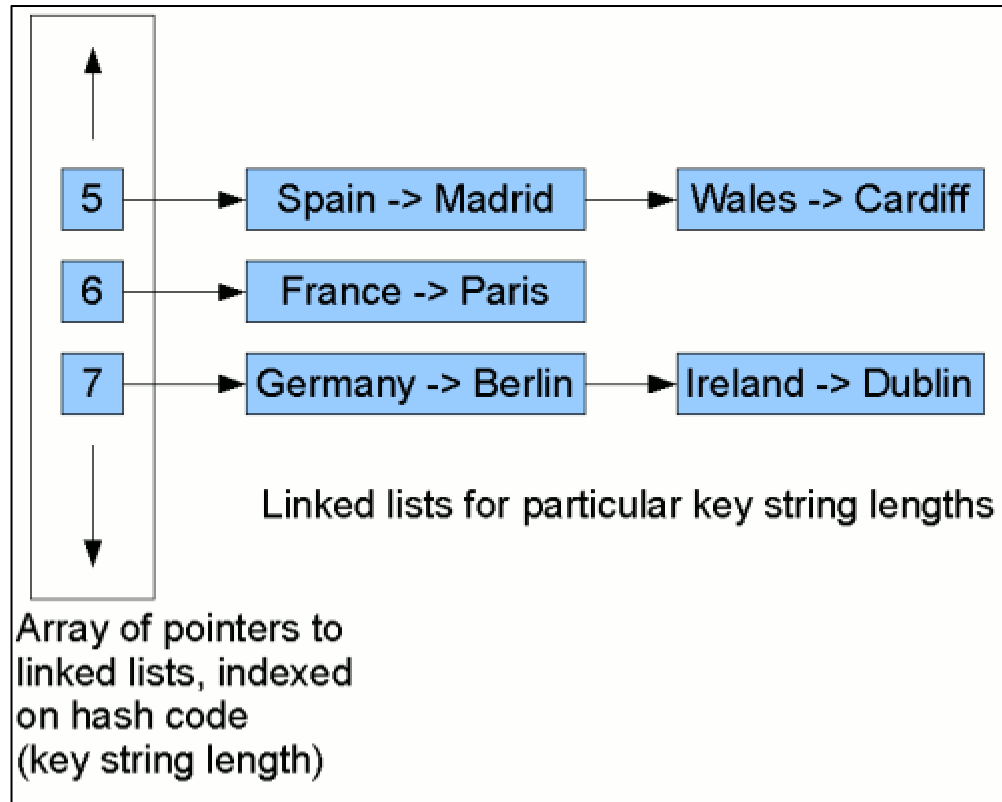
Position (hash code = key length)	Keys array	Values array
1		
2		
3		
4	Cuba	Havana
5	Spain	Madrid
6	France	Paris
7	England	London
8		
9		
10		
11	Switzerland	Berne

Solving possible collisions



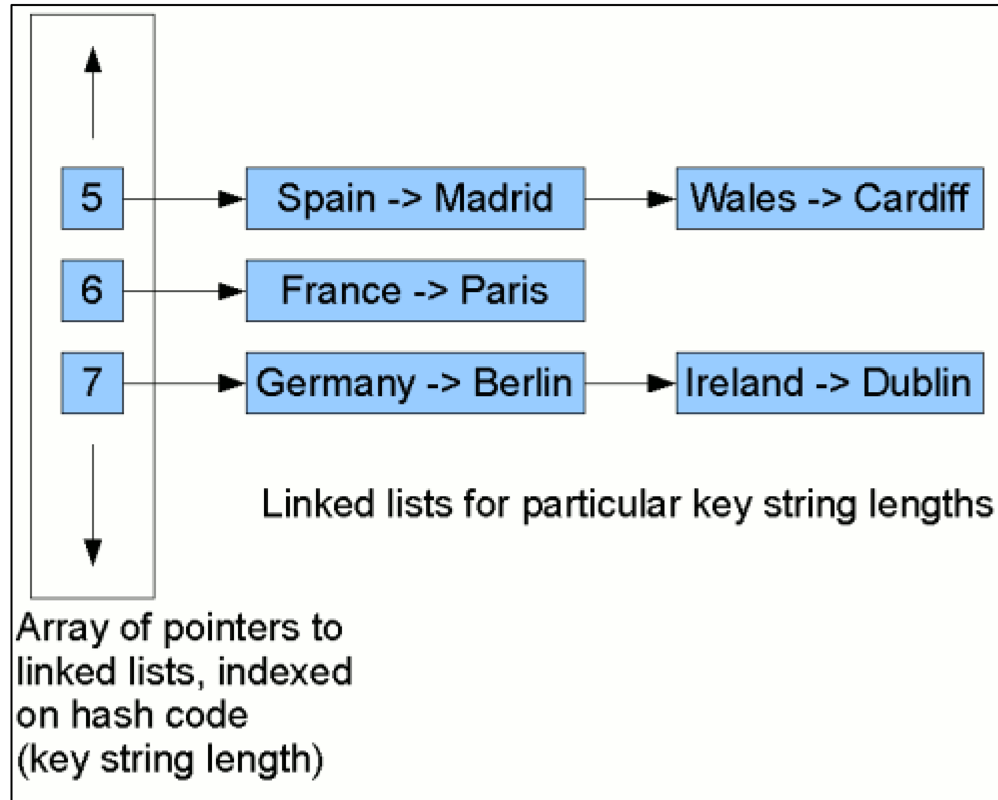
We can solve the problem of **collisions** by having an array of (references to) **linked lists** rather than simply an array of keys/values. Each little list is generally called a **bucket**.

Solving problem of very long strings



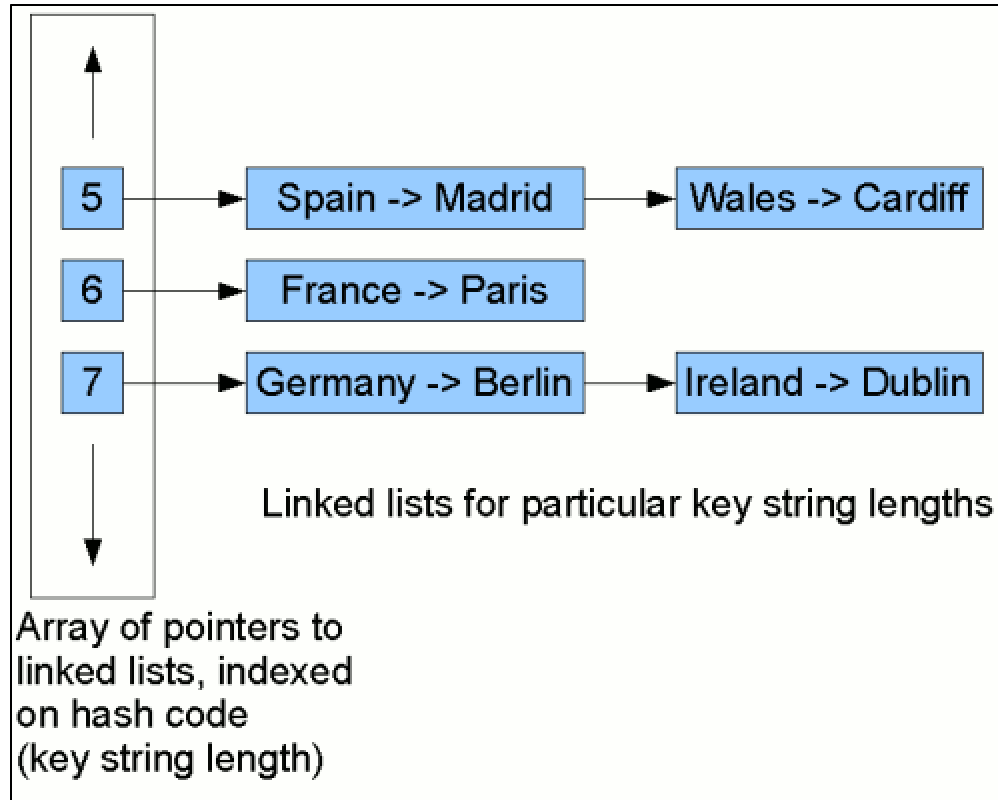
We can take care of too long values by taking modulo table size

Searching for a capital



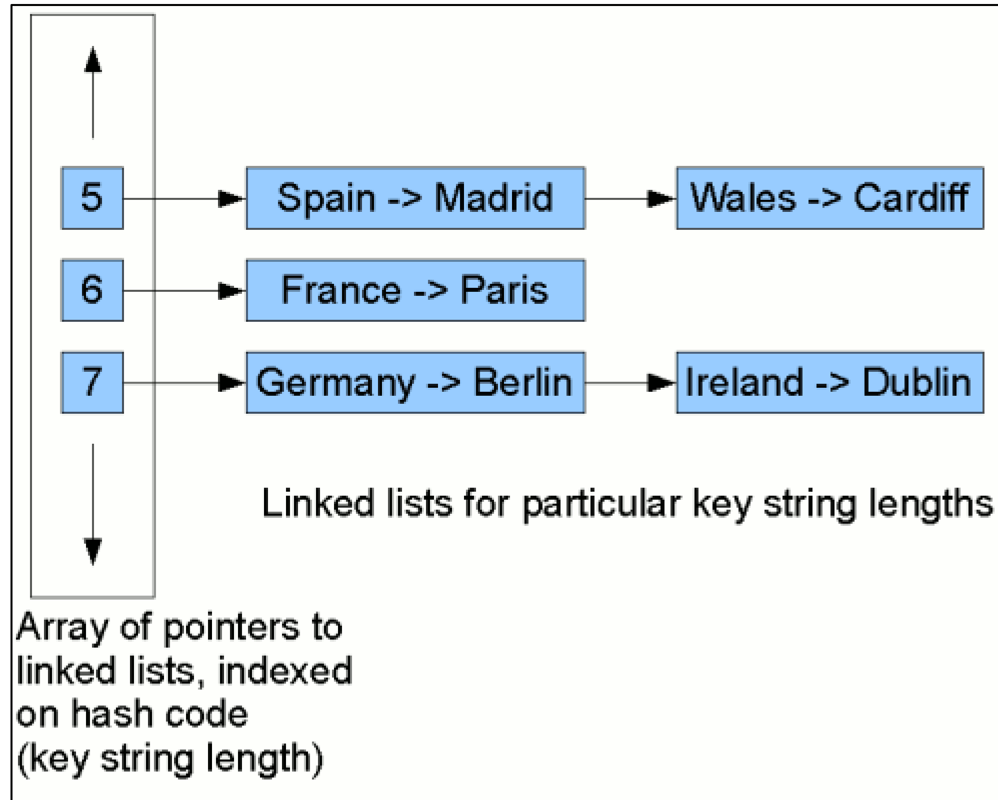
Each node in the linked lists stores a pairing of a key with a value. Now, to look for the mapping for, say, *Ireland*, we first compute this key's hash code (in this case, the string length, 7). Then we start traversing the linked list at position 7 in the table.

Searching for a capital



We traverse each node in the list, comparing the key stored in that node with *Ireland*. When we find a match, we return the *value* from the pair stored in that node (*Dublin*).

Searching for a capital



We find it on the second comparison. **If the list at a given position in the table is short**, we'll reduce significantly the amount of work we need to do to find a given key/value mapping.

Generic principle of a **good hash code**

A hash code that will cope with fairly "random typical" input and distribute the corresponding hash codes fairly randomly over the range of integers (32 bits in the case of Java)

That way the keys will be distributed reasonably evenly among the buckets.

Non-random distribution of bits in characters

Only low bits are distributed more or less randomly, bits 4 and 5 have a larger chance to be set to 1

Strings contain mostly numbers and lower case letters

097	0110 0001	a
098	0110 0010	b
099	0110 0011	c
100	0110 0100	d
101	0110 0101	e
102	0110 0110	f
103	0110 0111	g
104	0110 1000	h
105	0110 1001	i
106	0110 1010	j
107	0110 1011	k
108	0110 1100	l
109	0110 1101	m
110	0110 1110	n
111	0110 1111	o
112	0111 0000	p
113	0111 0001	q
114	0111 0010	r
115	0111 0011	s
116	0111 0100	t
117	0111 0101	u
118	0111 0110	v
119	0111 0111	w
120	0111 1000	x
121	0111 1001	y
122	0111 1010	z

048	0011 0000	0
049	0011 0001	1
050	0011 0010	2
051	0011 0011	3
052	0011 0100	4
053	0011 0101	5
054	0011 0110	6
055	0011 0111	7
056	0011 1000	8
057	0011 1001	9

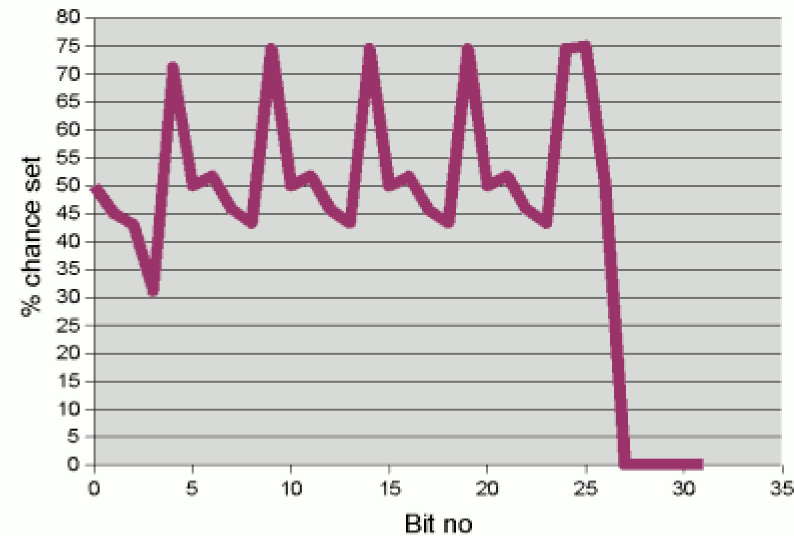
If we take a sum of all characters,

then we end up with numbers which have bit 4 or 5 set depending only on String length: odd/ even.

There would be no random distribution of high bits, and the high number of collisions will lead to inefficient search

Inducing randomness in non-random bits: first attempt

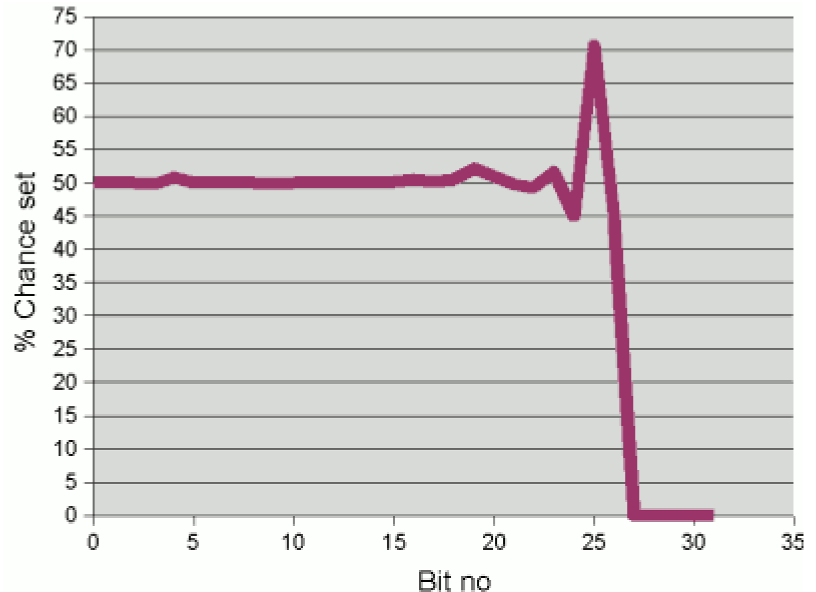
```
int hash = 0;
for (int i = 0; i < length(); i++) {
  hash = 32 * hash + charAt(i);
}
return hash;
```



Java String hashCode

Multiplying by **31** effectively means that we are shifting the hash by 5 places and then subtracting the original bits

```
int hash = 0;
for (int i = 0; i < length(); i++) {
  hash = (hash << 5) - hash + charAt(i);
}
return hash;
```



Recipe for spreading randomness over non-random bits

- Shift and sum

Java String hashCode: final version

```
public int hashCode()
{
      int hash = 0;
      for (int i = 0; i < length(); i++) {
            hash = hash*31+ charAt(i);
      }
      return hash;
}
```