CSE 373

Data Types and Manipulation; Arrays

slides created by Marty Stepp
http://www.cs.washington.edu/373/

Numeric data

type	kind of number	memory (bits)	range	examples
`byte`	integer	8	-128 .. 127	`(byte) 5`
`char`	character (integer)	16	\u0000 .. \uFFFF	`'a','\u2603'`
`float`	real number	32	-3.4e38 .. 3.4e38	`3.14f`, `1.1e7f`
`double`	real number	64	-1.8e308 .. 1.8e308	`3.14`, `6.022e23`
`int`	integer	32	-2^{31} .. 2^{31}	`42`, `-17`, `0xff`
`long`	integer	64	-2^{63} .. 2^{63}	`42L`, `21874109487L`
`short`	integer	16	-2^{15} .. 2^{15}	`(short) 42`

The Math class

Method name	Description
`Math.abs(`*value*`)`	absolute value
`Math.ceil(`*value*`)`	rounds up
`Math.floor(`*value*`)`	rounds down
`Math.log10(`*value*`)`	logarithm, base 10
`Math.max(`*value1*`, `*value2*`)`	larger of two values
`Math.min(`*value1*`, `*value2*`)`	smaller of two values
`Math.pow(`*base*`, `*exp*`)`	*base* to the *exp* power
`Math.random()`	random `double` between 0 and 1
`Math.round(`*value*`)`	nearest whole number
`Math.sqrt(`*value*`)`	square root

`Math.sin(`*value*`)`
`Math.cos(`*value*`)`
`Math.tan(`*value*`)`
	sine/cosine/tangent of an angle in radians

`Math.toDegrees(`*value*`)` `Math.toRadians(`*value*`)`	convert degrees to radians and back

Constant	Description
`Math.E`	2.7182818...
`Math.PI`	3.1415926...

Wrapper classes

- **wrapper**: an object whose sole purpose is to hold a primitive value.
  - often used with Java collections, since they can store only objects

Primitive Type	Wrapper Type
`boolean`	`Boolean`
`byte`	`Byte`
`char`	`Character`
`double`	`Double`
`float`	`Float`
`int`	`Integer`
`long`	`Long`
`short`	`Short`

- useful wrapper class behavior:
  - string -> primitive: `parse`*`Type`*
  - primitive -> wrapper: `valueOf`
  - wrapper -> primitive: *`type`*`Value`
  - `MIN_VALUE`, `MAX_VALUE`
  - Character: `isAlphabetic`, `isDigit`, `isLetter`, `isLowerCase`, `isUpperCase`, `isWhitespace`, `toLowerCase`, `toUpperCase`
  - Integer: `toString(int radix)`, `parseInt(`*`str`*`, `*`radix`*`)`

Interesting integers

- **factorial**: n! is the product of 1 .. n inclusive, e.g. 4! = 1*2*3*4=24

- **prime number**: only divisible by 1 and itself
  - **prime factorization**: e.g. 180 = 2*2*3*3*5
- **perfect number**: equals sum of its factors (e.g. 6=1+2+3)

- **greatest common divisor** (GCD)
  - Euclid's algorithm: GCD(a, b) = GCD(b, a mod b)
- **least common multiple** (LCM)
  - LCM(a, b) = (a*b) / GCD(a, b)

- **Fibonacci**: sum of the previous two: 1, 1, 2, 3, 5, 8, 13, 21, ...

BigInteger and BigDecimal

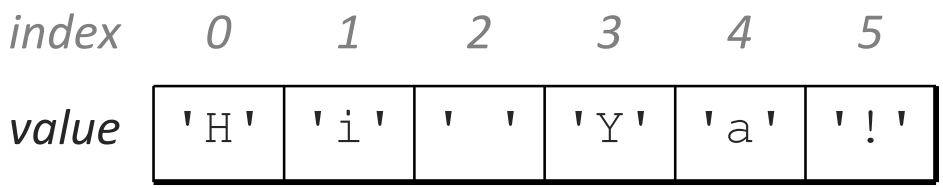
- can represent arbitrarily large integers and real numbers
  - `import java.math.*;`

`BigInteger(String)`	constructor
`abs()`	absolute value
`add(BigInteger)`, `divide(BigInteger)`, `multiply(BigInteger)`, `subtract(BigInteger)`	returns new big integer result
`equals(BigInteger)`, `compareTo(BigInteger)`	comparing two BigIntegers
`gcd(BigInteger)`	greatest common divisor
`isProbablePrime(`*`certainty`*`)`, `nextProbablePrime()`	checks whether integer is prime; gets next prime ≥ current integer
`pow(int exponent)`	exponentiation
`toString()`	BigInteger -> String
`valueOf(long)`	converts primitive into BigInteger

Text data: String and char

- **`String`** : A type of objects representing character sequences.
- **`char`** : A primitive type representing single characters.
  - A `String` is stored internally as an array of `char`

```
String s = "Hi Ya!";
```



  - It is legal to have variables, parameters, returns of type `char`
    - surrounded with apostrophes: `'a'` or `'4'` or `'\n'` or `'\''`

```
char initial = 'J';
System.out.println(initial);              // J
System.out.println(initial + " Joyce");   // J Joyce
```

String methods

Method name	Description
`contains(`**str**`)`	whether the given string is found within this one
`equals(`**str**`)`	whether two strings contain the same characters
`equalsIgnoreCase(`**str**`)`	whether two strings contain the same characters, ignoring upper vs. lower case
`indexOf(`**str**`)`	index where the start of the given string appears in this string (-1 if not found)
`length()`	number of characters in this string
`replace(`**a, b**`)`	replace all occurrences of string *a* with *b*
`split(`**separator**`)`	split string into array by the given separator text
`startsWith(`**str**`), endsWith`	whether it contains *str*'s characters at start/end
`substring(`**index1, index2**`)`	the characters from *index1* (inclusive) to *index2* (exclusive); if *index2* omitted, grabs till end of string
`toLowerCase(),toLowerCase`	a new string with all lower/uppercase letters

Comparing char values

- Get the characters of a string with the `charAt` method.
- You can compare `chars` with `==`, `!=`, and other operators:

```
String word = console.next();
char last = word.charAt(word.length() - 1);
if (last == 's') {
    System.out.println(word + " is plural.");
}

// prints the alphabet
for (char c = 'a'; c <= 'z'; c++) {
    System.out.print(c);
}
```

char vs. int

- Each `char` is mapped to an integer value internally
  - Called an **ASCII value**

    `'A'` is 65 `'B'` is 66 `' '` is 32

    `'a'` is 97 `'b'` is 98 `'*'` is 42

  - Mixing `char` and `int` causes automatic conversion to `int`.

    `'a' + 10` is 107, `'A' + 'A'` is 130

  - To convert an `int` into the equivalent `char`, type-cast it.

    `(char) ('a' + 2)` is `'c'`

char vs. String

- `"h"` is a `String`, but `'h'` is a `char` (they are different)
- A `String` is an object; it contains methods.

```
String s = "h";
s = s.toUpperCase();          // "H"
int len = s.length();         //  1
char first = s.charAt(0);     // 'H'
```

- A `char` is primitive; you can't call methods on it.

```
char c = 'h';
c = c.toUpperCase();                  // ERROR
s = s.charAt(0).toUpperCase();        // ERROR
```

  - What is `s + 1`? What is `c + 1`?
  - What is `s + s`? What is `c + c`?

Mutable StringBuilder

- A `StringBuilder` object holds a mutable array of characters:

method	description

`StringBuilder()`
`StringBuilder(`**capacity**`)`
`StringBuilder(`**text**`)`
	new mutable string, either empty or with given initial text

`append(`**value**`)`	appends text/value to string's end
`delete(`**start**`, `**end**`)` `deleteCharAt(`**index**`)`	removes character(s), sliding subsequent ones left to cover them
`replace(`**start**`, `**end**`, `**text**`)`	remove start-end and add text there
`charAt(`**index**`), indexOf(`**str**`), lastIndexOf(`**str**`), length(), substring(`**start**`, `**end**`)`	mirror of methods of String class
`public String` **`toString`**`()`	returns an equivalent normal String

String + implementation

- The following code runs surprisingly slowly. Why?

```
String s = "";
for (int i = 0; i < 40000; i++) {
    s += "x";
}
System.out.println(s);
```

- Internally, Java converts the loop into the following:

```
for (int i = 0; i < 40000; i++) {
    StringBuilder sb = new StringBuilder(s);
    sb.append("x");
    s = sb.toString();  // why is the code slow?
}
```

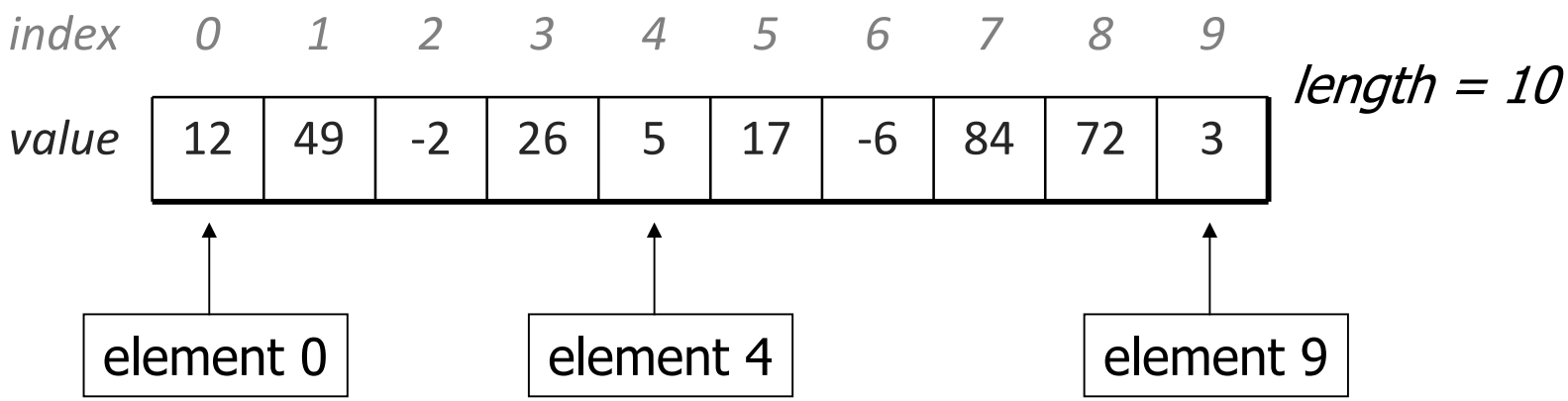
Tokenizing strings w/ Scanner

- `import java.util.*;`

Method	Description
`Scanner(String)`, `Scanner(File)`, `Scanner(InputStream)`	constructor
`next()`	reads a one-word `String` from the input
`nextLine()`	reads a one-*line* `String` from the input
`nextInt()`, `nextDouble()`	reads an `int`/`double` from the input and returns it
`hasNext()`	returns `true` if there is a next token
`hasNextLine()`	returns `true` if there are any more lines of input
`hasNextInt()`, `hasNextDouble()`	returns `true` if there is a next token and it can be read as an `int`/`double`

Arrays

- **array**: An object that stores many values of the same type.
  - **element**: One value in an array.
  - **index**: A 0-based integer to access an element from an array.
  - **length**: Number of elements in the array.

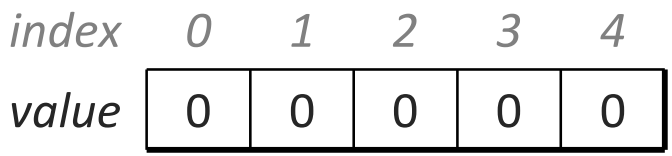


Array declaration

type`[]` **name** `=` `new` **type**`[`**length**`];`

- Length explicitly provided. All elements' values initially 0.

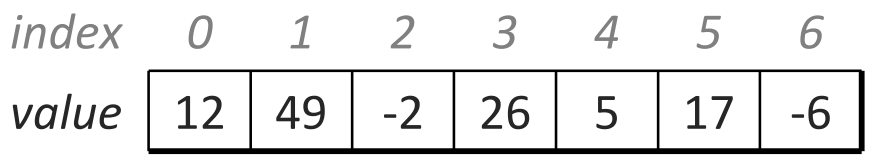
```
int[] numbers = new int[5];
```



type`[]` **name** `=` `{`**value**`,` **value**`,` … **value**`};`

- Infers length from number of values provided. Example:

```
int[] numbers = {12, 49, -2, 26, 5, 17, -6};
```



Accessing elements; length

```
name[index]                    // access
name[index] = value;           // modify
name.length
```

- Legal indexes: between **0** and the **array's length - 1**.

```
numbers[3] = 88;
for (int i = 0; i < numbers.length; i++) {
    System.out.print(numbers[i] + " ");
}
System.out.println(numbers[-1]);  // exception
System.out.println(numbers[7]);   // exception
```

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*
value	12	49	-2	**88**	5	17	-6

The `Arrays` class

- Class `Arrays` in package `java.util` has useful static methods for manipulating arrays:

Method name	Description
`binarySearch(`**array, value**`)`	returns the index of the given value in a sorted array (< 0 if not found)
`copyOf(`**array, length**`)`	returns a new array with same elements
`equals(`**array1, array2**`)`	returns `true` if the two arrays contain the same elements in the same order
`fill(`**array, value**`)`	sets every element in the array to have the given value
`sort(`**array**`)`	arranges the elements in the array into ascending order
`toString(`**array**`)`	returns a string representing the array, such as `"[10, 30, 17]"`