CS 112 – Introduction to Computing II

Wayne Snyder
Computer Science Department
Boston University

Today: Java basics:

- Compilation vs Interpretation
- Program structure
- Statements
- Values
- Variables
- Types
- Operators and Expressions

Next Time: Java Statements, conditionals, and loops

Reading assignments will be posted on the web site!

Computer Science

Compilation vs Interpretation

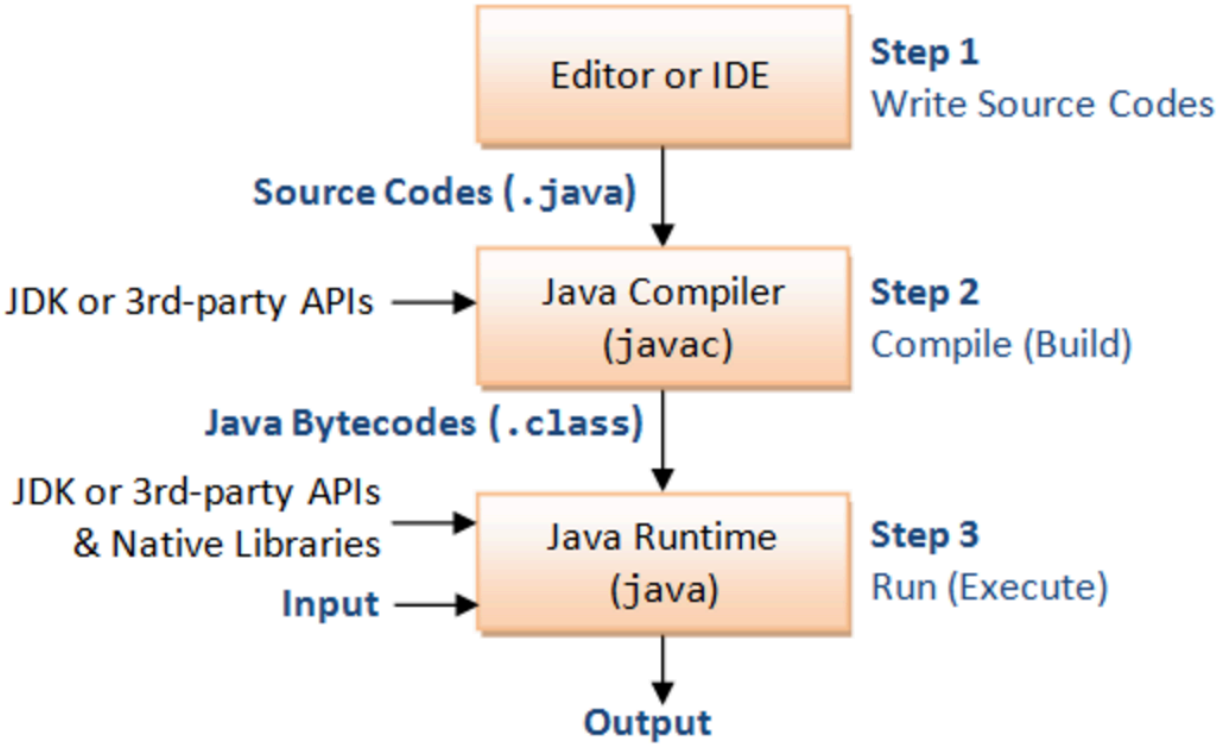
Python is an example of an interpreted language; the primary workflow is to interact with the interpreter as a fancy calculator with lots of features:

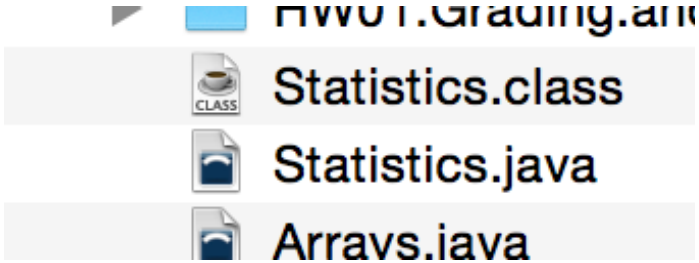
```
739 #    Xe.append(nextExponential(50))
740 #
741 #Xn = []
742 #
743 #for i in range(N):
744 #    Xn.append(norm.rvs(50,10))
745
746
747
748 #drawNormalPlot(Xu)
749 #
750 #drawNormalPlot(Xp)
751 #
752 #drawNormalPlot(Xe)
753 #
754 #drawNormalPlot(Xn)
755
756 # Normality testing by binning
757 # Separate data into bins of width S * sigma on each side of mea
758 # and graph percentages of data that are within
759
760
761 def drawNormal(mu, var):
762     plt.figure(figsize=(15,7))
763
764     sigma = var**0.5
765
766     Xn = [x for x in np.arange(int(mu-4*sigma),int(mu+4*sigma)+1
767     Yn = [phi(mu,var,x) for x in Xn]
768     plt.plot(Xn,Yn, 'r-')
769     plt.xlabel('Range')
770     plt.ylabel('Probability')
771     plt.xlim(int(mu-4*sigma),int(mu+4*sigma))
772 #     plt.title(r'N())
773
```

```
24994    67.21126
24995    69.50215
24996    64.54826
24997    64.69855
24998    67.52918
24999    68.87761
Name: Height, dtype: float64

In [106]: H.var()**0.5
Out[106]: 1.9016787712056042

In [107]: 3_4
  File "<ipython-input-107-c8029b172030>", line 1
    3_4
     ^
SyntaxError: invalid syntax


In [108]: 3 + 5
Out[108]: 8

In [109]: 2 + 9
Out[109]: 11

In [110]: H.var() ** 0.5
Out[110]: 1.9016787712056042

In [111]:
```

Compilation vs Interpretation

Java is an example of a language which is compiled; before running any code, your program (ending in .java) must be transformed into a lower-level form (an "executable file" ending in .class), and which is then passed to the interpreter, which runs the program and produces output:





Java Basic Program Structure

Java programs are organized as classes (more on these next week!) stored in files with the suffix ".java", and with code written inside methods delimited by curly braces; each program must have a method called main, which contains the code that will be executed when you run your program:

Class name is same as file name

SampleProgram.java

Curly braces enclose classes and methods

```
public class SampleProgram {

   public static void main(String[] args) {

         //  Here is code that is executed when
         //  you run your program.

   }

}
```

Class

Method

Java Comments

Java has comments, exactly like Python, but with a different syntax:

Python:

""" and #

```
"""
File: Lab05.py
Author: Wayne Snyder
Purpose:  This collects together
discrete distributions.


"""

#  Import statements

import matplotlib.pyplot as plt
import matplotlib.mlab as mlab
import numpy as np
import math
```

Java:

/* .... */ and //

```
/*
   File: Statistics.java
   Author: Wayne Snyder
   Date: January 23rd, 2015
   Purpose: This is a solution for
 */

// The following is a library whic
// reading input from the user in
// libraries (such as Math) are al
// as Scanner) you need to explici
// must occur before your class de
```

Java Statements

In Python, we compute by evaluating expressions, which yield a value, which is then printed out.

In Java, we compute by executing statements, which have an effect on the state of the program: they assign a value to a variable, or print a value out, or otherwise change something in the system.

Python:

```
24994    67.21126
24995    69.50215
24996    64.54826
24997    64.69855
24998    67.52918
24999    68.87761
Name: Height, dtype: float64

In [106]: H.var()**0.5
Out[106]: 1.9016787712056042

In [107]: 3_4
  File "<ipython-input-107-c8029b172030>", line 1
    3_4
     ^
SyntaxError: invalid syntax


In [108]: 3 + 5
Out[108]: 8

In [109]: 2 + 9
Out[109]: 11

In [110]: H.var() ** 0.5
Out[110]: 1.9016787712056042

In [111]:
```

Java:

```
public class SampleProgram {

   public static void main(String[] args) {

         int x = 4;
         int y = 6;
         int z = x + y;
         System.out.println( z );

   }

}
```

Assignment Statements

Note: All Java statements end in semicolon.

It is often useful to understand the effect of a sequence of assignment statements by tracing the values of the variables, which change after each statement. The collection of all values is the state of the program:

	a	b	t
`int a, b;`	*undefined*	*undefined*	
`a = 1234;`	1234	*undefined*	
`b = 99;`	1234	99	
`int t = a;`	1234	99	1234
`a = b;`	99	99	1234
`b = t;`	99	1234	1234

Java Values and Types

A Data Type (or just Type) is a collection of values and associated operations.

Java is a Strongly-Typed language supporting many different types:

Java Primitive Data Types				
Type	**Values**	**Default**	**Size**	**Range**
byte	signed integers	0	8 bits	-128 to 127
short	signed integers	0	16 bits	-32768 to 32767
int	signed integers	0	32 bits	-2147483648 to 2147483647
long	signed integers	0	64 bits	-9223372036854775808 to 9223372036854775807
float	IEEE 754 floating point	0.0	32 bits	+/-1.4E-45 to +/-3.4028235E+38, +/-infinity, +/-0, NAN
double	IEEE 754 floating point	0.0	64 bits	+/-4.9E-324 to +/-1.7976931348623157E+308, +/-infinity, +/-0, NaN
char	Unicode character	\u0000	16 bits	\u0000 to \uFFFF
boolean	`true, false`	`false`	1 bit used in 32 bit integer	NA

String "hi there" ""

Java Values and Types

However, in CS 112 we will only use the following types:

type	set of values	literal values	operations
`char`	characters	`'A'` `'@'`	compare
`String`	sequences of characters	`"Hello World"` `"126 is fun"`	concatenate
`int`	integers	`17` `12345`	add, subtract, multiply, divide
`double`	floating-point numbers	`3.1415` `6.022e23`	add, subtract, multiply, divide
`boolean`	truth values	`true` `false`	and, or, not

Java Values and Types

Literal values are similar to Python:

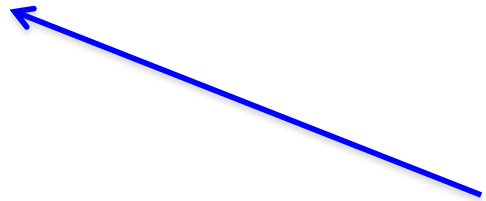
int 4 -5

double 3.4 -2.34e10

char ‘a’ ‘\n’ ‘\t’ // single quotes for chars

boolean true false // note lower case

String “hi there” // must use double quotes



Note that String is capitalized

Java Values and Types

Python is “weakly typed”: values have types but variables do not; variables are just names for any value you want and can be reused for any values; the only errors occur when variables have not yet been assigned values:

```
In [123]: X = 5

In [124]: X
Out[124]: 5

In [125]: X = 4.5

In [126]: X = "hi"

In [127]: X
Out[127]: 'hi'

In [128]: Z
Traceback (most recent call last):

  File "<ipython-input-128-41ff0912a07f>", line 1, in <module>
    Z

NameError: name 'Z' is not defined
```

Java is strongly-typed in that

All variables must be declared with a type before being used and can only be used for that type of value:

```
int x;                      // declare x to be int
x = 4;                      // assign value to x
System.out.print(x);
double y = 3.4;             // combine declaration and assignment
double z = x + y;
System.out.print(z);
```

During compilation, the types are checked and errors will be reported with line numbers and terse explanations:

This might seem unduly rigid, but the philosophy of strongly-typed languages is that specifying types makes programmers more careful about variables, and bugs and errors can be found during compilation, not when the program is running.

Values can be converted from one type to another implicitly or explicitly:

Widening Conversions (implicit):

Narrow types (less information) ⟶ Wider types (more information)

Example: `int` ⟶ `double`

```
double  x;
x  = 4;   // 4 is widened to 4.0 and then assigned
```

No error!

This might seem unduly rigid, but the philosophy of strongly-typed languages is that specifying types makes programmers more careful about variables, and bugs and errors can be found during compilation, not when the program is running.

Values can be converted from one type to another implicitly or explicitly:

Widening Conversions (implicit):

Narrow types (less information) ⟶ Wider types (more information)

Example: `int` ⟶ `double`

```
double  x;
x  = 4;   // 4 is widened to 4.0 and then assigned
```

Example 2: `char` ⟶ `int`

```
int  x;
x  = 'A';   //  'A' is converted to its Unicode
            //  value 65 and assigned to x
```

This might seem unduly rigid, but the philosophy of strongly-typed languages is that specifying types makes programmers more careful about variables, and bugs and errors can be found during compilation, not when the program is running.

Values can be converted from one type to another implicitly or explicitly:

Narrowing Conversions (you must specify a cast or else get an error):

Wider types (more information) ⟶ Narrower types (less information)

Example: `double` ⟶ `int`

```
int  x;
x  = 4.5;
```

Error!

This might seem unduly rigid, but the philosophy of strongly-typed languages is that specifying types makes programmers more careful about variables, and bugs and errors can be found during compilation, not when the program is running.

Values can be converted from one type to another implicitly or explicitly:

Narrowing Conversions (you must specify or else get an error):

Wider types (more information) ⟶ Narrower types (less information)

Example: `double` ⟶ `int`

```
int  x;
x  = 4.5;
```

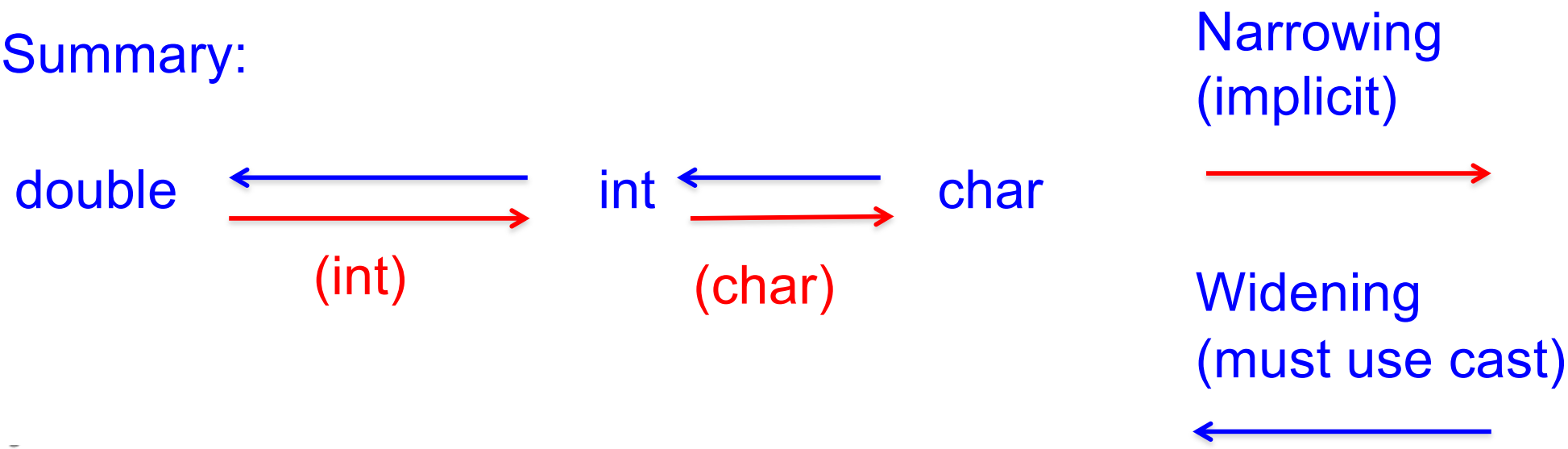
Error!

Must explicitly tell Java to truncate the double value to an integer:

```
int  x;
x  = (int) 4.5;     // x gets 4
```

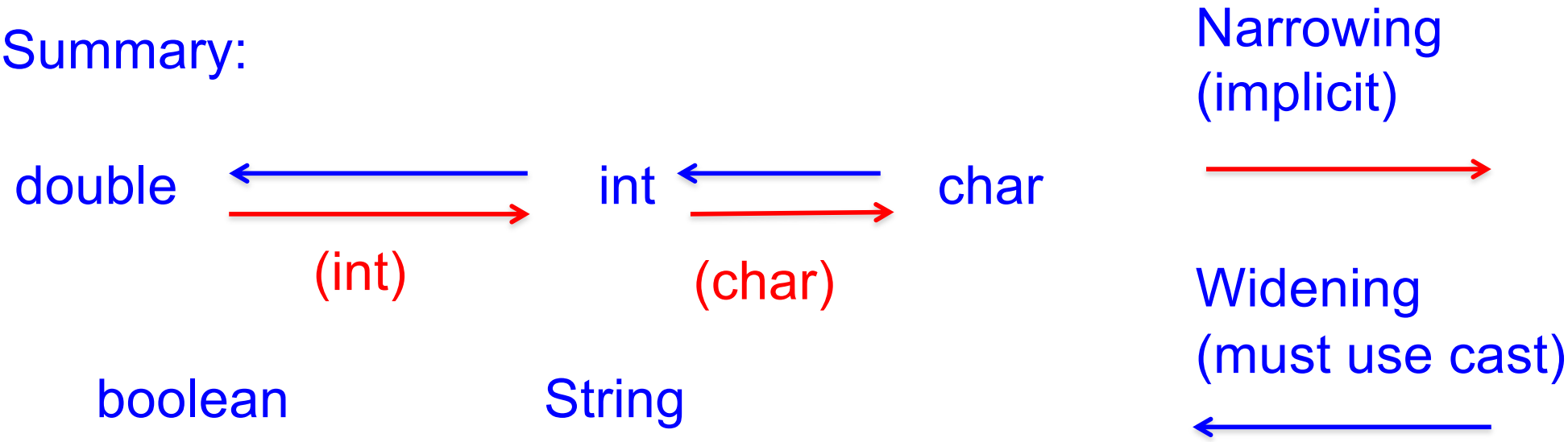
Cast

Summary:




```
10  public static void main(String[] args) {
11
12    double x = 0.0;
13    int i    = 48;
14    char c   = 'a';
15
16    x = i;          // widening conversion int -> double
17    i = c;          // widening conversion char -> int
18    x = c;          // widening conversion char -> double
19
20    i = x;          // error!
21    i = (int) x;    // fixed with explicit cast double -> int, now i == 48
22    c = i;          // error!
23    c = (char) i;   // fixed with explicit cast int -> char, now c = '0' (ASCII of '0' == 48)
24
```

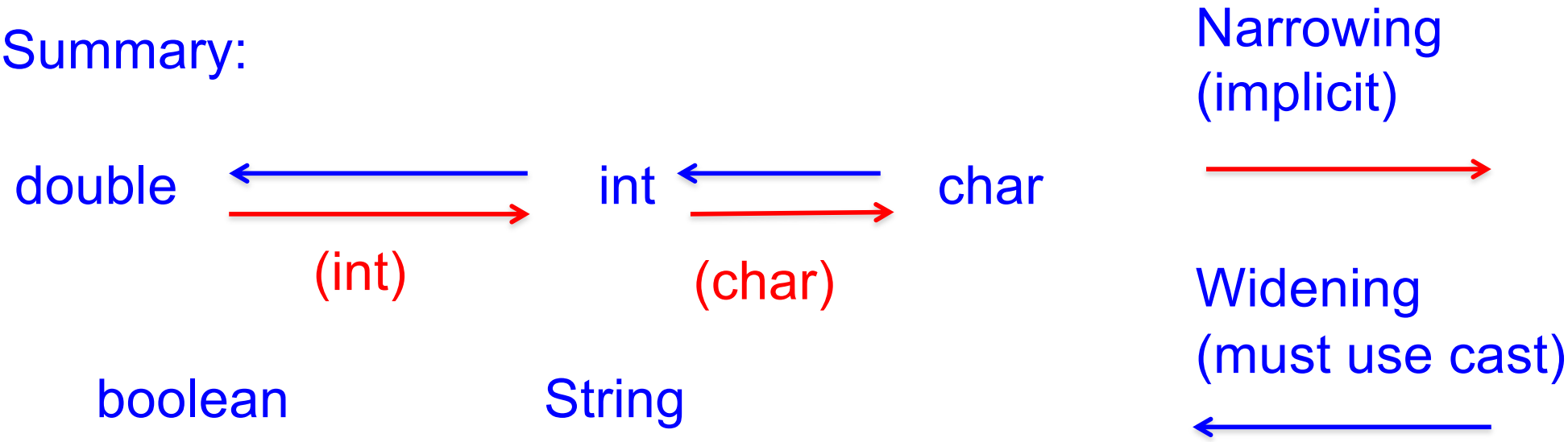
Summary:




```
10  public static void main(String[] args) {
11
12    double x = 0.0;
13    int i    = 48;
14    char c   = 'a';
15    boolean b = false;
16    String s = "hi there";
17
18    i = b;
19    b = x;
20    s = c;
21    s = i;
22    s = b;
23
```

Strings and booleans are incompatible for conversions – must find a work-around!

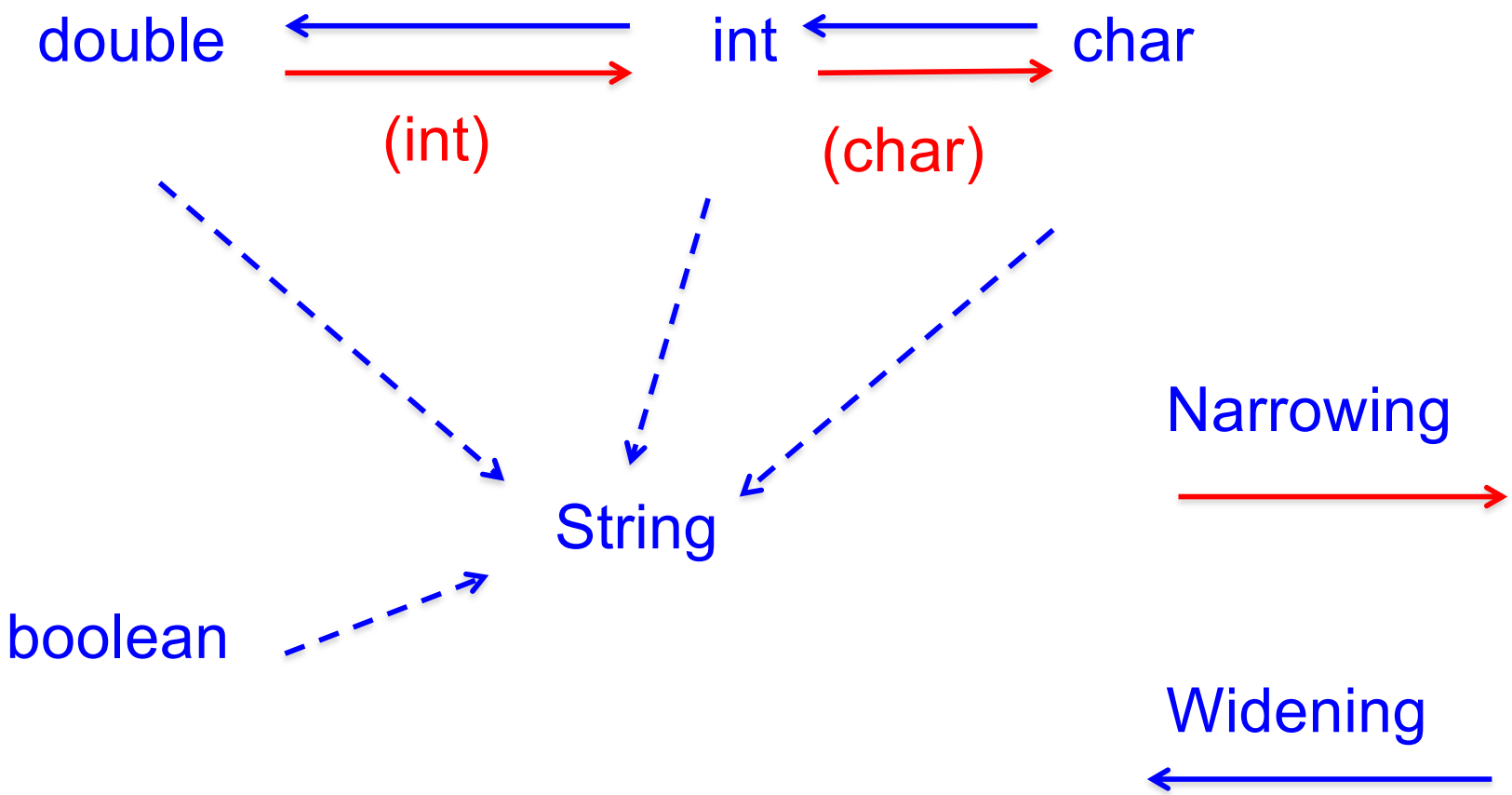
Summary:




```
10  public static void main(String[] args) {
11
12    double x = 0.0;
13    int i    = 48;
14    char c   = 'a';
15    boolean b = false;
16    String s = "hi there";
17
18    i = b;
19    b = x;
20    s = c;
21    s = i;
22    s = b;
23
```

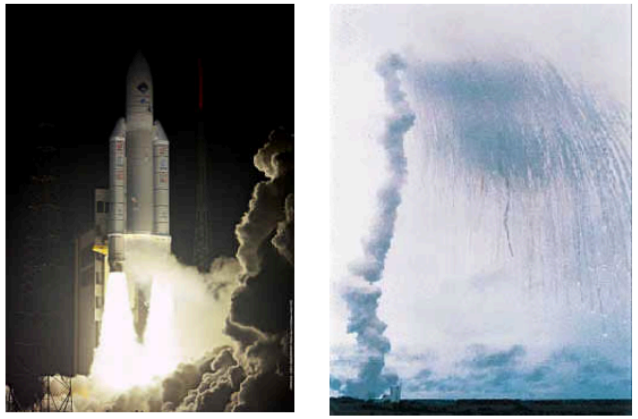
Strings and booleans are incompatible for conversions – must find a work-around!

The Workaround: All the types have methods for turning their values into Strings:


double
int
char
(int)
(char)
String
boolean
Narrowing
Widening

Digression: You probably think this is a purely academic matter, and making a type conversion mistake will only lose you a few points on the midterm….

Think again: In 1996, the Adriade 6 rocket exploded after takeoff because of a bad type conversion in its control code:



example of bad type conversion

Java Operators

The operators are almost exactly the same as in Python:

Same:

+	addition	+=
-	subtraction	-=
*	multiplication	*=
%	modulus	%=
==	equals	
!=	not equal	
<	less	
<=	less or equal	
>	greater	
>=	greater or equal	

When Java evaluates an overloaded operator, it automatically performs widening conversions as necessary to make the operands fit the operator:

Example: + is overloaded – it works for two ints or two doubles….

```
public static void main(String[] args) {

    int x = 4;

    double y = 2.3;

    System.out.println( ( x + x ) );     // prints:   8

    System.out.println( ( y + y ) );     // prints:   4.6

    System.out.println( ( x + y ) );     // prints:   6.3

}
```

All the arithmetic operators in java are overloaded for int and double.

Widening Conversion:

4 + 2.3

4.0 + 2.3 => 6.3

Result is the wider type!

Division is overloaded, but behaves differently for ints and doubles…..

Python: two different division operators:

/ floating-point division /=

// integer division

Java: division operator is “overloaded”:

/ returns an int if both operands are ints, otherwise returns double:

5 / 2 => 2 5.0 / 2 => 2.5 5.0 / 2.0 => 2.5

5 / (double) 2 => 2.5

Java Operators

The boolean operators in Java look different (although they work exactly the same):

Python:	Java:
not	!
and	&&
or	||

Note that in both languages, and and or are lazy:

```
(false && X)   =>  false (without evaluating X)
(true  || X)   =>  true (without evaluating X)
```

Example:

```
( (4 < 6) && ( 5 >= 5) )  =>  true      // both < and >= are evaluated
( (7 < 6) && ( 5 >= 5) )  =>  false     // only < needs to be evaluated
```

Java Operators

There is NO exponentiation operator in Java:

Python:

```
x  ** 2        x squared
```

Java: have to use explicit math functions:

```
Math.pow(x,2)    =>   returns double
```

You will become familiar with the explicit Math functions in HW 01.

Finally, Java has several useful increment and decrement operators which Python lacks; these can be used as statements OR expressions:

Statements:

```
++x;    x++ ;        // same as x = x + 1     or    x += 1
--x;    x-- ;        //  same as x = x – 1    or    x -=  1
```

Expressions:

```
++x       has the value AFTER adding 1
x++       has the value BEFORE adding 1
```

	x	y	z
int x = 4;	4	undef	undef
int y = ++x;	5	5	undef
int z = x++ ;	6	5	5

The char data type is useful when we manipulate Strings (which are simply sequnces of chars. Here are the most useful methods in the Character library:

Useful Character library methods (the first three of these return boolean values; read about here)

Note: these methods are called using the "dot notation" but with the name of the library, NOT the name of a variable. The first two will not be used in t
them, since we will surely use them soon!

```
Character.isLetter( c )          // returns true or false depending on whether the character c is a letter

      Character.isLetter( 'A' ) => true
      Character.isLetter( '9' ) => false

Character.isDigit( c )

Character.isWhitespace( c )      // returns true if character is defined as whitespace on your computer

Character.toString( c )           // converts a character to a String

       String s = Character.toString( 'A' );     // s will have the value "A"
                                                 // Think of this as typecasting a char to a String
```

Note: You can also just use the Character in a String context, e.g., System.out.println("Hi" + ' ' + "There!)

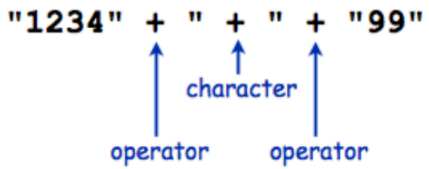
Java Operators

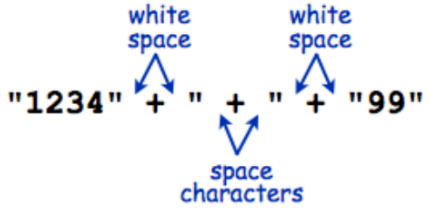
The String data type is necessary for manipulating textual data and for input and output:

values	sequences of characters
typical literals	`"Hello,"` `"1 "` `" * "`
operation	concatenate
operator	+

expression	*value*
`"Hi, " + "Bob"`	`"Hi, Bob"`
`"1" + " 2 " + "1"`	`"1 2 1"`
`"1234" + " + " + "99"`	`"1234 + 99"`
`"1234" + "99"`	`"123499"`

Caveat. Meaning of characters depends on context.





Note that + is overloaded: it can be used for plus (int, double) or concatenate (Strings).

Java String Data Type


```
String str = "Hi There";
```

`charAt( n )` -- returns the character at the given index n in the string (starts at 0):

```
str.charAt(1) =>'i'
```

NOTE: You CAN NOT change a String by doing for example:

```
str.charAt(1) = 'h'
```

Strings are immutable, and to change a String you have to create a new one!

`toLowerCase()` -- converts all letters to lower case;

```
str.toLowerCase() => "hi there"
```

```
str = str.toLowerCase(); // this is how you convert the string str to lower case
```

equals(str2) -- Returns `true` or `false` depending on whether the string is equal to the parameter

```
str.equals("hi There") => false
```

```
str.equals("Hi There") => true
```

Note: DO NOT use == to compare Strings, this is almost always the wrong thing to do!

Java String Data Type

compareTo(str2) -- Compares the string with another string using the lexicographic order (dictionary ordering) and returns an integer: 0 if equal, negative if less, and positive greater:

```
str.compareTo("Hi There") => 0
```

```
str.compareTo("String") => negative number, since "Hi There" < "String"
```

```
str.compareTo("Hi The") => positive number, since "Hi There" > "Hi The"
```

```
str.compareTo("CS112") => positive number, since "Hi There" > "CS112"
```

NOTE on how to use compareTo(....): compare the result with 0; the way you compare the result to 0 is the same as the comparison you want to do:

```
(str.compareTo("Hi There") == 0) => true
```

```
(str.compareTo("Hi!") == 0) => false
```

```
(str.compareTo("String") < 0) => true, since "Hi There" < "String"
```

```
(str.compareTo("Hi The") > 0) => true, since "Hi There" > "Hi The"
```

```
(str.compareTo("CS112") < 0) => false, since is not true that "Hi There" < "CS112"
```

Punchline:

(str.compareTo(str2) R 0) is equivalent to (str R str2)

where R is one of ==, !=, <, >, <= , >=

Java String Data Type

`replace(...)` -- Create a new String where one substring is replaced by another -- the original String is unchanged.

```
str.replace("Hi", "Hello") => "Hello There"

str.replace("e", "") => "Hi Thr"

str = str.replace("e", ""); // remove all occurrences of the character 'e' from str
```

`length()` -- Return the length (number of chars in) the String. NOTE that this is different from the way you find the length of an array: `A.length`

```
str.length() => 8
```

`split("\\s+")` -- Separate each String into separate strings using the white space between them as a separator and put them in an array (here we are giving parameter which does this, others can be used as well).

```
String[] words = str.split("\\s+");

// words is now the array ["Hi", "There"]
```