Java Data Types

The concept of data types

- The main objective of a program is to manipulate various data: whole numbers, real numbers, characters, words,…

  `numberOfEmployees = 5 … 20 … 300 –` *can be only a whole number*

  `temperature = -12.5 … 0.0 … 36.6 … 95.4 –` *can be a real number*

  `grade = ‘F’ … ‘P’ … ‘C’ … ‘D’ –` *can only be a character*

  `doorIsLocked = yes … no –` *can only be “yes” or “no”*

  `city = Canberra … London … New York` – *can be a word, or two words*

  - Numbers can be added, multiplied, or subtracted, but how can these operations be extended to characters and words?

  - Algebra defines simple rules how numbers can be compared, but how to compare words or sentences?

  - How to define and store complex data such as contact details, timetable, weather, …?

  - How can this diversity of values be efficiently stored in memory and correctly processed?

The concept of data types

- Generic programming languages must efficiently process all types of data and therefore must efficiently describe various types of data
- Java language requires that all variables be given a data type
- Data types determine:
  - what values are legal
  - how much memory is allocated to store a variable
  - what kind of operations are allowed
- Java supports eight fundamental data types and a mechanism of defining application specific data types
- Selection of appropriate data types is the first step in developing efficient software applications

Fundamental data types

byte	8 bits	very small whole numbers	Min value = -128 Max value = 127
short	16 bits	whole numbers	Min value = -32768 Max value = 32767
int	32 bits	big whole numbers	Min value = -2147483648 Max value = 2147483647
long	64 bits	very big whole numbers	Min value = -9223…5808 Max value = 9223…5807
float	32 bits	single-precision real numbers	$\pm 3.40282347\times 10^{38}$
double	64 bits	double-precision real numbers	$\pm 1.797693134862\times 10^{308}$
boolean	1 bit	true or false	false or true
char	16 bits	characters	Unicode characters coded from 0 to 65535

A fatal error: Ariane 5

https://www.youtube.com/watch?v=kYUrqdUyEpI

Cause: *data type range error*

Assignment of a 64-bit double value to a 16-bit integer value representing horizontal bias caused an operand error, because the double value was too large to be represented by a 16-bit integer, leading to the 1996 crash 37 seconds after launch

https://en.wikipedia.org/wiki/Ariane_5

Default data types

- Although Java has four data types (`byte`, `short`, `int`, `long`) to represent integer numbers there is no any performance advantage in using `byte`, or `short`
- All arithmetic operations are carried out by JVM with `int` precision. **The default type for integers should be `int`**
- `byte`, or `short` are mostly useful when you need to save memory, or in some special cases (to be discussed later)
- There is no any performance advantage in using `float` instead of `double`. Math methods work with `double` type
- Java compiler uses `double` as a default type to represent real numbers. **Use `double` by default for real numbers**

```
double x = 7.35;   // OK because 7.35 is double by default
float y = 7.35;    // compilation error: loss of precision
float y = 7.35F;   // OK, 7.35 is explicitly set to float
```

Data types of literals

- When you declare a variable you specify its data type. The specified data type reflects all properties of the variable.
- When you use a literal in your program, how can the compiler guess what its data type is

  `23` – is it `int` or `long?`

  `12.75` – is it `float` or `double?`

- To avoid confusion, you should attach **suffixes** to literals

```
float y = 4.37F;  // F indicates float data type
long a = 37654L;  // L indicates long data type
```

- In some cases you may need to attach **prefixes** to literals

```
int x = 101;        // 101 is a decimal value
int y = 0b101;      // 0b is a prefix for binary values
                    // 0b101 is equal to 5 in decimal
```

Arithmetic operators

- Although basic arithmetic operators are defined using the same set of symbols (`+ - * /`) their operation is a bit different for integer and floating-point types

  Example:

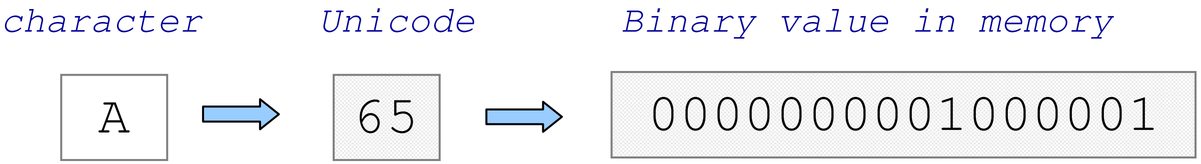
  ```
  int a = 7, b= 2, c;
  c = a / b;   // this is integer division: c = 3

  float x = 7.0F, y= 2.0F, z;
  z = x / y;   // this is floating point division: z = 3.5
  ```

- Integer division by zero terminates further program execution, while floating-point division by zero only results in a value INFINITY
- Arithmetic operations (`+` or `-`) on `char` data also have different meaning

char data type

- All numeric values are stored in the program memory as signed binary numbers
- Text characters are not numbers

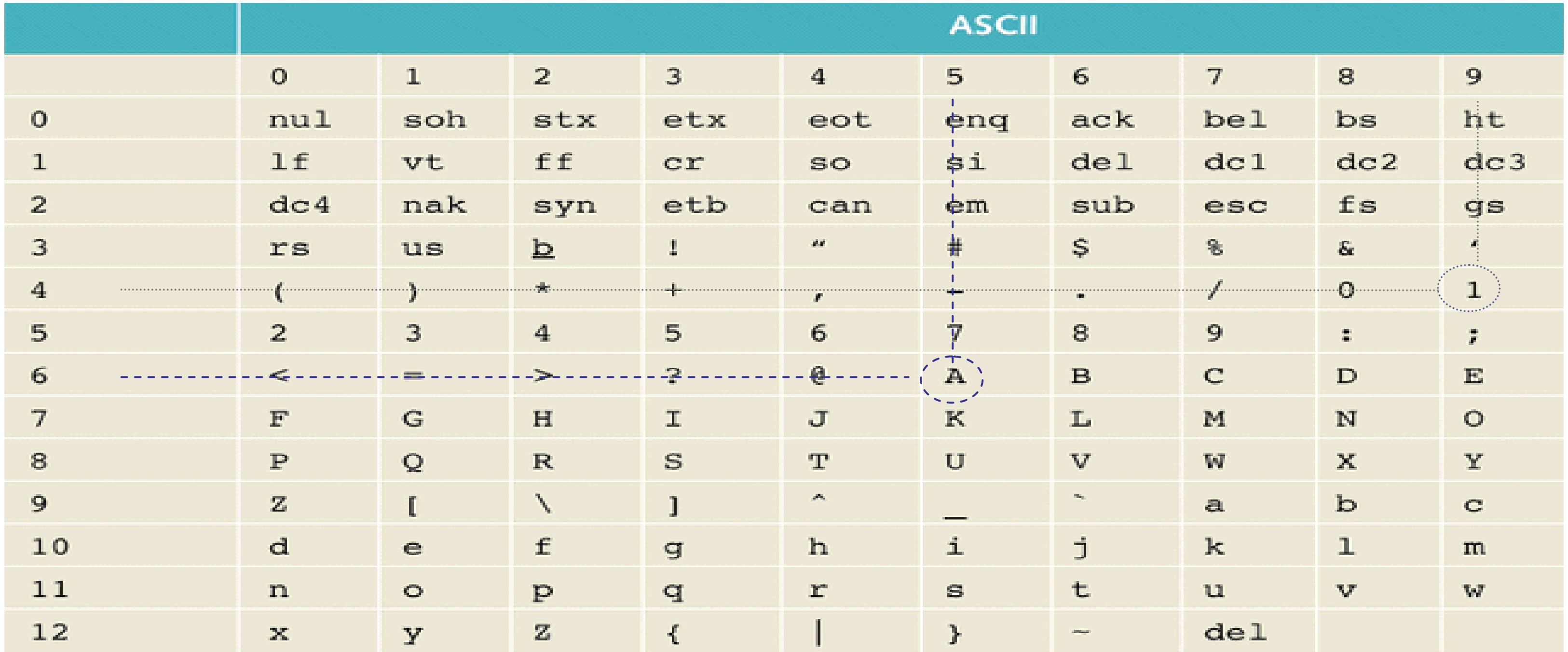
  `char prefix = 'D';` <- *How is the character* ***D*** *stored in memory?*
- Java uses Unicode to represent characters. There are 65536 codes (from 0 to 65535). Each character is assigned with a unique code


character *Unicode* *Binary value in memory*

A → 65 → 0000000001000001

- The Unicode Standard supports 125 character sets for various languages
- First 128 codes, which are known as ASCII codes, are allocated for Latin characters and print control symbols

ASCII Codes



	ASCII									
	0	1	2	3	4	5	6	7	8	9
0	nul	soh	stx	etx	eot	enq	ack	bel	bs	ht
1	lf	vt	ff	cr	so	si	del	dc1	dc2	dc3
2	dc4	nak	syn	etb	can	em	sub	esc	fs	gs
3	rs	us	b̲	!	"	#	$	%	&	'
4	(	)	*	+	,	-	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[	\	]	^	_	`	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{	|	}	~	del		

Example: ASCII code for ‘A’ is 65
ASCII code for ‘1’ is 49

Special ASCII codes

‘\b’ – backspace (bs) `'\t'` – horizontal tab (ht) `'\n'` – new line (lf)

‘\r’ – return (cr) ‘\v’ – vertical tab (vt) ‘\0’ – null (nul)

char data type

- Declaration of char variables

```
char newCharacter;  // only declaration
char grade = 'P';   // declaration and initialization
char level = 65;    // declar. and initialization to 'A'
```

- You can use arithmetic expressions with literals to assign to `char` variables

```
 newCharacter = 'A' + 2;  // assigned with 'C'
```

Such expressions are processed at the compilation time. The compiler is smart enough to pre-compute `'A' + 2` as 'C', then `newCharacter = 'C'` will be executed by JVM at run time

- You cannot use arithmetic expressions where at least one of the operands is a `char` type variable

```
 newCharacter = grade + 15;  //error: loss of precision
```

Here the compiler cannot pre-compute a value as the variable `grade` can store any Unicode at run time

Expressions and statements

Programmers must understand how statements and expressions specified in Java programs are processed (by `javac` and `JVM`)

```
int a=5, b;

b = a + 2;
```

JVM at run time

1. *A copy of the lvalue* `a` *is created as an rvalue*
2. `rvalue+rvalue` *produces an rvalue of type int*
3. *The produced rvalue is assigned then to the lvalue* `b`

```
final double CT = 4.0;
double x=5.0, y;

y = CT + 5.0;
```

Java compiler

1. *The compiler pre-computes CT + 5.0 into another rvalue 9.0*
2. *A statement* `y= 9.0;` *is actually compiled*

JVM at run time

The rvalue 9.0 is assigned to lvalue y

```
x = CT + x * 2.0;
```

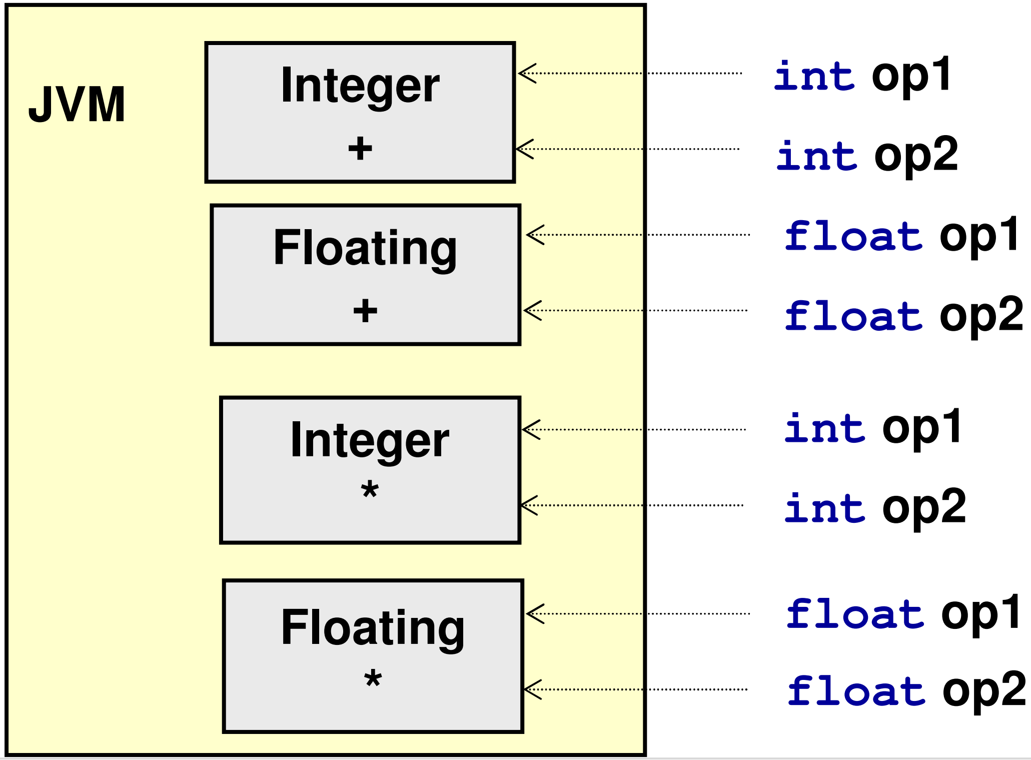
JVM at run time

1. *A copy of the lvalue* `x` *is created as an rvalue*
2. `rvalue * rvalue` *produces an rvalue of type* `double`
3. `rvalue + rvalue` *produces an rvalue of type* `double`
4. *The produced rvalue is assigned then to the lvalue* `x`

Mixed type expressions

What is the data type of the rvalue produced by an expression if it contains operands of different data types?

Example: `2 * 12.25` - how is it processed by JVM ?



Operands op1 and op2 **must have the same data type** before they are sent to JVM computation modules: adders, multipliers, etc

Data type conversion

- Operands of different types **must** be converted to a common data type before they can be processed by JVM
  - Widening : conversion to a data type with higher precision
  - Narrowing: conversion to a data type with lower precision
- Java compiler does widening conversion **automatically**

```
wage = 2 * 14.8;    /* 2 is auto converted to 2.0 */
```

- You can **explicitly** convert a value to any data type (widening or narrowing) depending on your needs

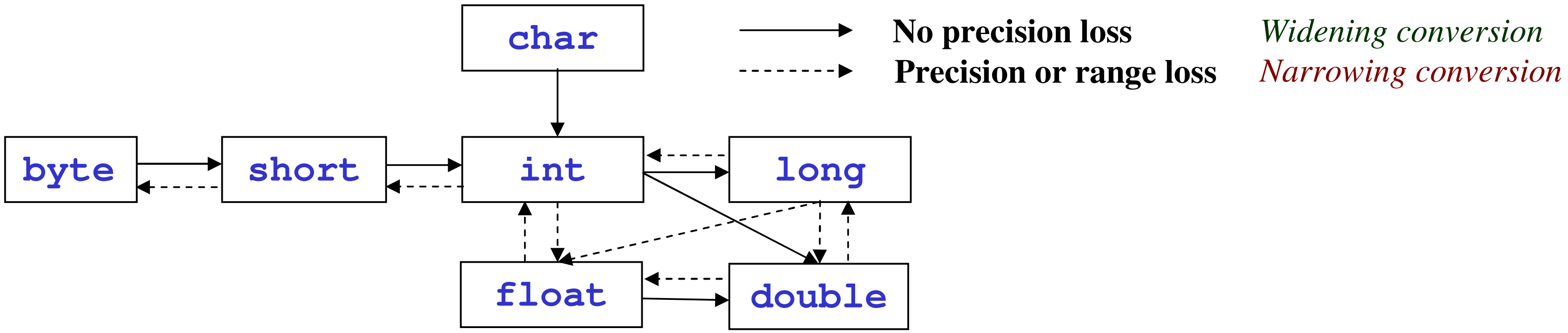
Syntax:

```
        (cast_type)expression;
```

Examples:

```
        (int)12.8;          // 12.8 is converted to int 12
        (float)length;      // length is converted to float
```

Conversion between numeric types



char

byte short int long

float double

No precision loss *Widening conversion*

Precision or range loss *Narrowing conversion*

- Auto conversion is carried out between compatible types according to a set of predefined rules (widening conversion)
- If auto conversion is not successful → compiler error

```
int n1 = 65, n2 = -65;
double db = n1;        // OK, db = 65.0

int n = 123456666;
float f = n;      // Error, f can only be 1.23456667e8  (precision loss)
```

A large integer may have more digits than the float type can represent

- You can use explicit type cast where loss of precision is acceptable

```
float measuredSpeed = 60.75F;            // narrow 60.75 to a float
int  carSpeed = (int)measuredSpeed;      // carSpeed = 60
char ch = (char)n1;                      // ch = 'A' (65 is Unicode of A)
char ct = (char)n2;                      // ? (-65 is not a Unicode)
```

Expressions with auto conversion

```
int a = 1, b = 2;
short sa = 1, sb = 2;
long la = 1L, lb = 2L;
float fa = 1.0F, fb = 2.0F;
double da = 1.0, db = 2.0;

a = a  -  sa; // short sa is auto converted to an int rvalue
               // then int = int - int ( OK )

b = a + la;    // int a is auto converted to a long rvalue
               // int = long + long error: loss of precision

db = a * da;    // int a is auto converted to a double rvalue
                // double = double * double

fb = da  + fa;  // float is auto converted to double
                // float = double + double error: loss of precision
```

Expressions with auto conversion

- Sometimes even simple arithmetic operations may be confusing

Example:

```
byte a = 5, b;
b = a + 1;    <- ???
```

It should work, but this statement results in a compilation error

`error: possible loss of precision` ← *Why?*

- According to the Java Language Specification
  - if operands of an arithmetic expression have types `byte` or `short`, they are automatically promoted to `int`

```
  byte a = 5, b;
byte     int    +   int
✗ b =   a   +   1;
✓ b = (byte)(a + 1);
```

- As the arithmetic operations are carried out with at least `int` precision, it may not make sense to use `byte` or `short` data

- Use explicit type conversion if you have to process `byte` or `short` data

Quiz

What values are assigned to `result ?`

```
int a = 5, b = 6;
double fa=5.0, fb=6.0, result;

result = (a + b)/2;                /* result = 5.0 */
result = (a + b)/2.0;              /* result = 5.5 */
result = (fa + b)/2;               /* result = 5.5 */
result = (double)(a + b)/2;        /* result = 5.5 */
result = (double)((a + b)/2);      /* result = 5.0 */
```

Overflow and precision limits

- Eight fundamental Java data types allows the programmers to process efficiently a wide spectrum of data. However…

1. Even careful selection of data types for your program doesn't guarantee that the result will always be right

Example:

```
  double result = 10.0/3.0;   // 3.33333333…
```

Even the most accurate data type may not have precision sufficient for some numbers

2. JVM doesn't check if integer values go over the range

Example:

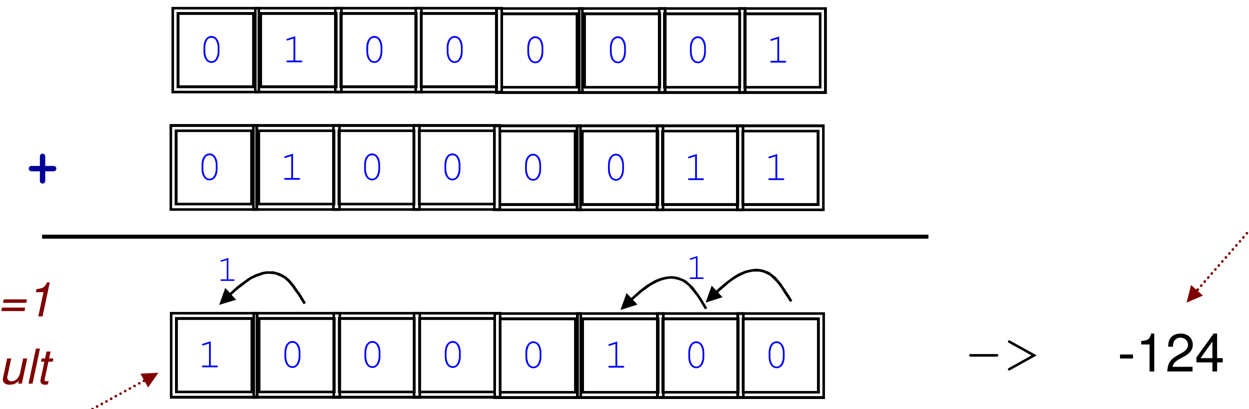
```
byte x = 129;   // compilation error
                // javac checks the range when possible
int y1 = 129, y2 = -127;
byte z = (byte)y2;      // No error, but z = -127
z = (byte)( y2 - 2);  // No error, but z =  127
```

Overflow

- Arithmetic expressions may produce incorrect values when data bits propagate into the sign bit

Example:

```
byte result, num1 = 65, num2=67;
result = (byte)(num1 + num2);
print( result );  // is it 132 ?
```



0 1 0 0 0 0 0 1

+ 0 1 0 0 0 0 1 1

1 0 0 0 0 1 0 0 -> -124

No errors reported

A carry bit =1 turns the result into negative

Arithmetic operators

- Besides `+ – * /,` Java has 40 other built-in operators
- Remainder %

  Example: You have 43 tires in stock. A client usually needs a set of 4 new tires. How many tires will be left presuming that all clients order 4 tires?

  ```
  int tiersLeft = inStock % 4;  // the remainder is 3
  ```

  Remainder is a leftover after integer division
- Remainder is defined for floating numbers too

  Example: You have 21.5 kilograms of baking flour. Each day you use 1.7 kilograms of flour. How many times does 1.7 go into 21.5 and how much will be left over?

  ```
  double leftOver = 21.5 % 1.7;  // the remainder is 1.1
  ```

Arithmetic operators

Operations *x = x + 1* and *x = x – 1* are used so frequently that Java defines special operators for them

- Increment operator ++ (increase by 1)
  - Pre-increment *Example:* `++counter`
    increment happens before the value to be used
  - Post-increment *Example:* `counter++`
    increment happens after the value is used
- Decrement operator -- (decrease by 1)
  - Pre- decrement *Example:* `--counter`
    decrement happens before the value to be used
  - Post- decrement *Example:* `counter--`
    decrement happens after the value is used

Increment and decrement can be applied only to an lvalue:

~~`(counter + 3)`~~`--`

Pre-increment and post-increment

expressions	**example** **(assume** `sum` **is** `10`**,** `counter` **is** `5` **)**
`counter++;`	counter = 6
`++counter;`	counter = 6
`sum = sum + counter++;`	sum = 15 counter = 6
`sum = ++counter + sum;`	sum = 16 counter = 6

Equivalent to:

```
counter = counter +1;

sum = counter + sum;
```

Equivalent to:

```
sum = sum + counter;

counter = counter +1;
```

Compound assignment operator

assignment	compound assignment
`sum = sum + number;`	`sum` **`+=`** `number;`
`product = product * number;`	`product` **`*=`** `number;`

Syntax:

variable **op=** *expression;*

Meaning:

variable **=** *variable* ***op*** *expression*

Examples:

```
count += 2;             // count = count + 2;
sum += 2+3;             // sum = sum + (2+3)
stock -= quantity;      // stock = stock - quantity
power *= 2.71;          // power = power * 2.71
div /= power+20.0;      // div = div/(power+20.0)
rem %= d;               // rem = rem % d
```

Evaluation of complex expressions

Calculate the value of the following expression

```
-2 * -3/(4%5 + 6) + 4
```

Rules used in Java for evaluation of expressions

1. Precedence

   which operators (+, *, /, %, …) are evaluated first

2. Associativity

   how an operator is associated with operands

   `5-2-1` is it `(5-2)-1 = 2` ?

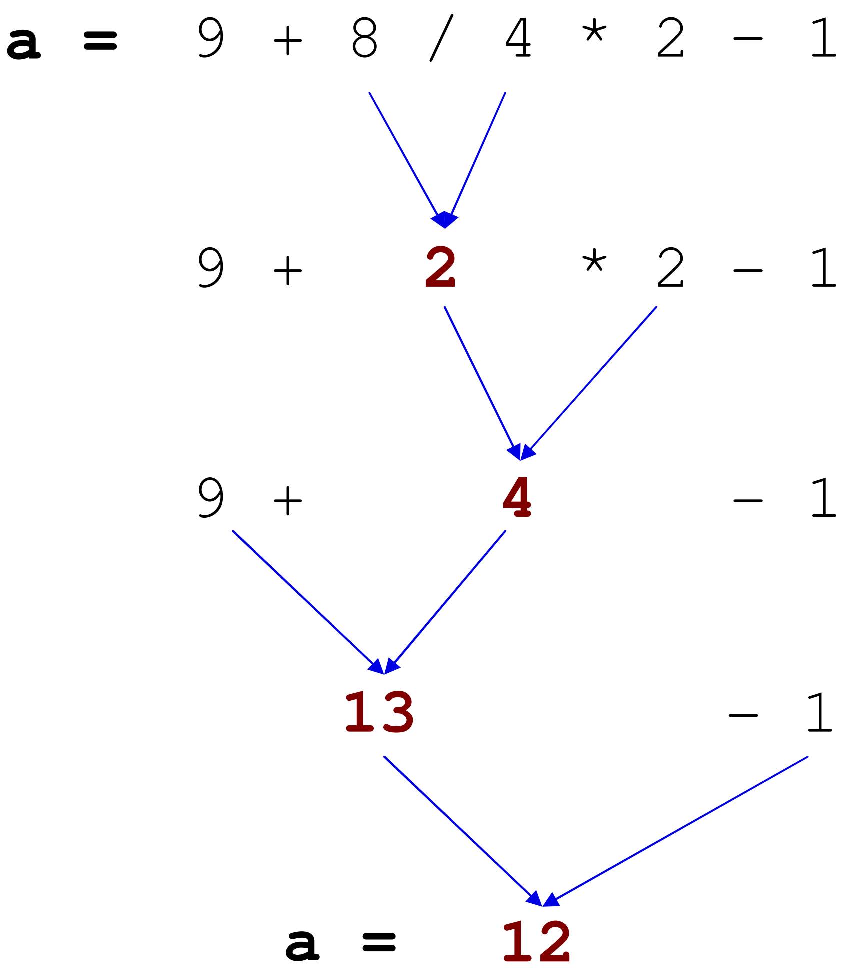
   or `5-(2-1) = 4` ?

Arithmetic operator precedence

Operators	Associativity
function calls , (operations in brackets)	left to right
pre++, pre--, unary -, (type_cast)	right to left
*, /, %	left to right
+ , -	left to right
= , += , -= , *= , /= , %=	right to left

```
Examples:

result = (15-6) * (3+11);

result = (float)++day * 2.5;

result *= 15 - day + 5;

result += b = c;
```

Example: precedence



a = 9 + 8 / 4 * 2 – 1

Division first as it's on the left

9 + 2 * 2 – 1

9 + 4 – 1

Addition first as it's on the left

13 – 1

a = 12

Quiz

Which expressions are not implemented correctly?

$\dfrac{ab}{a+b}$	↔	`a * b / ( a + b )`	
$a+\dfrac{b}{c^2}$	↔	`a + b / c * c`	X
$\dfrac{\dfrac{a}{b}+c}{a-\dfrac{b}{c}}$	↔	`a/b + c/( a - b/c )`	X
$c=p(1+r)^y$	↔	`c = p * pow( (1+r), y )`	

Bitwise operators

- Java provides operators that act separately upon individual bits which comprise a value
- The bit-wise operators can be used only on integral data types `byte, short, int, long`

1. Invert bits ~

```
byte a = 127;        // 01111111
a = (byte)~a;        // 10000000  -128 in decimal
```

2. Left shift <<

```
byte a = 3;          // 00000011
a = (byte)(a<<2);    // 00001100  (12 in decimal)
```

3. Right shift >>

```
byte a = 64;         // 01000000
a = (byte)(a>>2);    // 00001000  (8 in decimal)
```

`enum` data type

- Some programs require variables which can take only a limited number of predefined values. Can `int` type provide a solution?

Example:

```
int today = 1;       // SUNDAY
today = 7;           // SATURDAY
today = 10;          // ? (out of the expected range)
today = -1;          // ? (out of the expected range)
```

- You can define a new data type that has a few possible values

Day is a new data type

```
class Example
{
   public enum Day {MONDAY, TUESDAY, WEDNESDAY, FRIDAY}

   public static void main(String[] args)  {
       Day lectureDay;                // a variable of type Day
       lectureDay = Day.MONDAY;         // OK
       lectureDay = Day.SUNDAY;         // error
       lectureDay = Day.FRIDAY;         // OK
       System.out.print(lectureDay);    // output: FRIDAY
   }
}
```

Class as a user defined data type

- A variable of a basic data type can describe only one property

```
double weight;    // describes weight
double price;     // describes price
Paint colour;     // describes colour of the paint
int numOfDoors;   // describes the number of doors
```

Is any data type that could combine several related properties into a collection

- Class can be considered as a programmer-defined data type that can contain a collection of properties (fields) and behaviours (methods)

```
class Car {
  private double weight;    // describes weight
  private double price;     // describes price
  private Paint colour;     // describes the paint colour
  private int numOfDoors;   // describes the number of doors
}
```

- You can declare variables of defined class types

```
Car bmwI8; // a variable of the type Car
```

Reference variables

- Although declarations of basic type variables and variables of type class may look similar

```
int numberOrdered;  // a variable of type int
Car bmwI8;          // a variable of the type Car
```

there is a fundamental difference between them

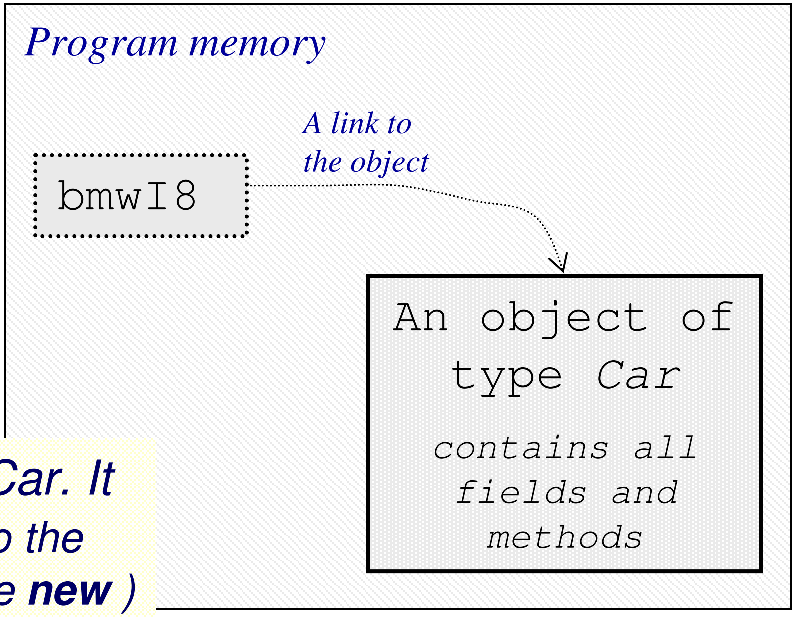
- Classes are complex data types. As a result memory allocation for them is a two stage process:

1. A variable must be declared

```
Car bmwI8;
```

2. An object must be created explicitly

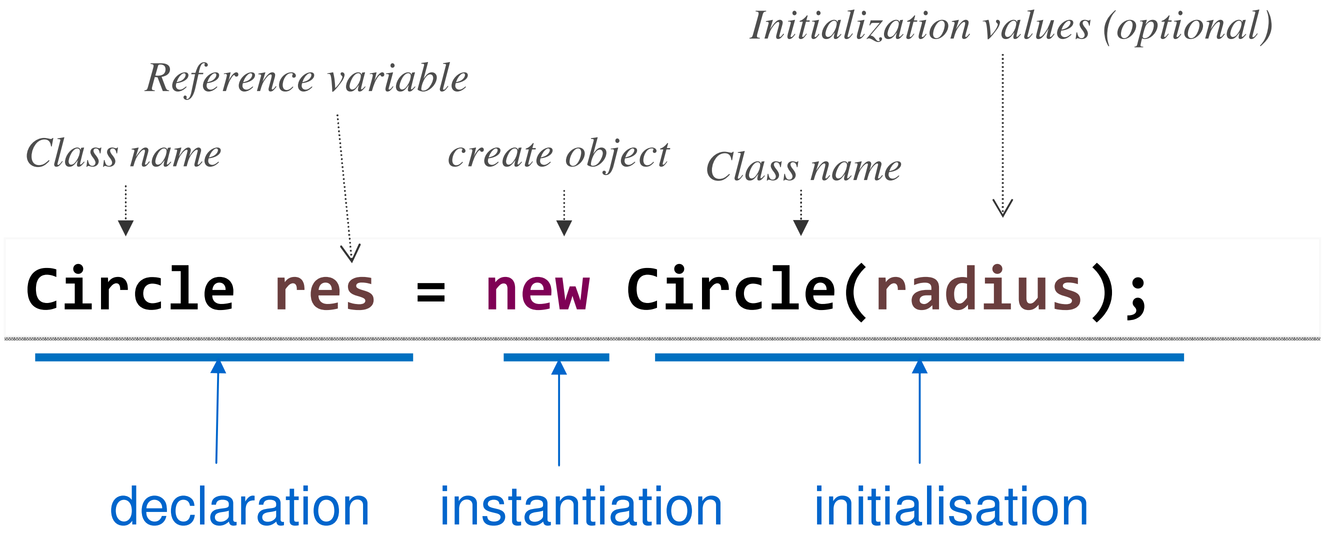
```
bmwI8 = new Car();
```



*bmwI8 is not an object of class Car. It is a **reference variable** (linked to the object that must be created using the **new**)*

Creating objects

- Software objects – created from a class
  - State – stored in **fields**
  - Related behaviour – implemented through **methods**



Initialization values (optional)

Reference variable

Class name

create object

Class name

```
Circle res = new Circle(radius);
```

declaration

instantiation

initialisation

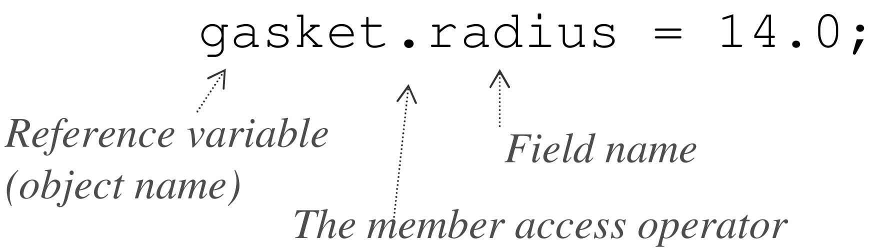
Using objects

- Once an object has been created,

```
Circle gasket = new Circle( 2.5 );
```

it can be used

1. You can access its public fields directly



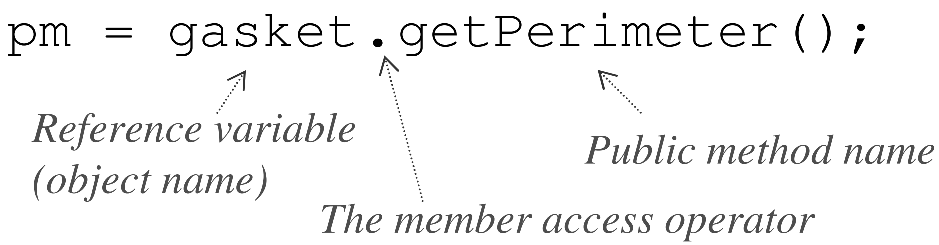
```
gasket.radius = 14.0;
```

Reference variable (object name)

Field name

The member access operator

2. You can access its private fields only through public methods



```
pm = gasket.getPerimeter();
```

Reference variable (object name)

Public method name

The member access operator

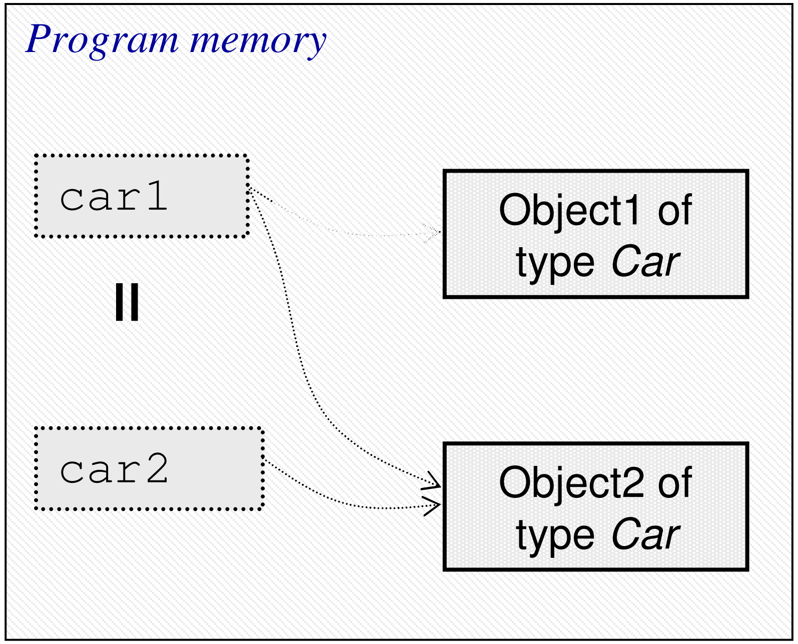
Reference variables

- There is a substantial difference between basic type and reference type variables:

```
int a = 1, b = 5;
a = b;    // a and b are equal to 5
b = 4;    // b is 4, a is 5
```

Variables `a` and `b always` have different locations in memory

```
Car car1 = new Car();
Car car2 = new Car();

car1 = car2;
```



- *When you assign a reference variable to another one, only the link is copied*
- *`Object2` is referenced by both variables*
- *`Object1` is lost in memory*

Garbage collection

Can ghost objects resulted from wrong operations with reference variables fill all memory space?

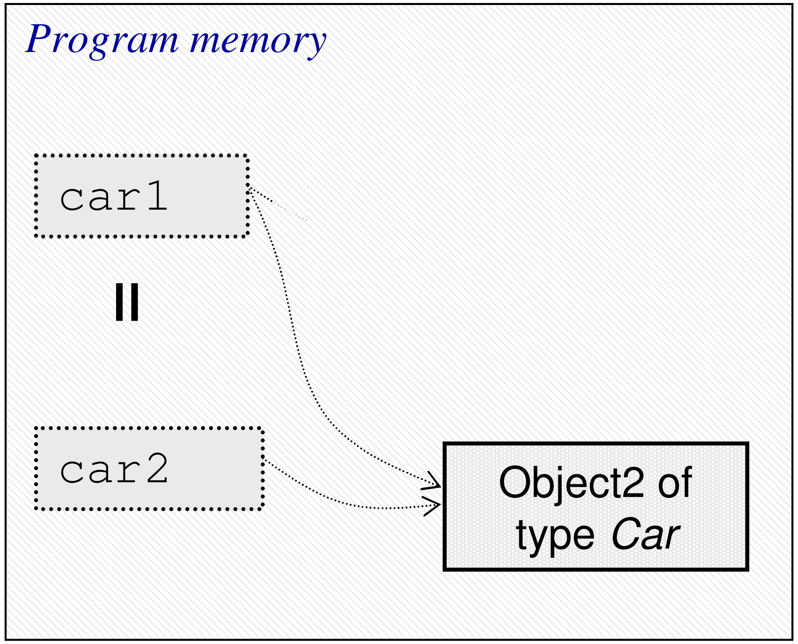
- JVM uses a technique called Garbage Collection to automatically detect and delete those objects which are no longer in use

You need to understand how to use reference variables properly to avoid generation of ghost objects:

-Generation of 'ghost' objects indicates that you may have bugs in your code and your application may not work correctly

- Automatic garbage collection consumes additional JVM resources

- It can start at any time and slow down (even pause) your program



Strings

- Many applications need to process textual information
  - text editors
  - word processors
  - messaging tools
  - command windows
- Character-by-character text processing that uses char data type is not practical
- In computer science, a sequence of characters is called a string
- Java provides the `String` class to create and manipulate strings. It is defined in `java.lang` package.

```
// 1. Create a reference variable of type String
String str1;
// 2. Create an object of type String linked to the variable
str1 = new String("A string is a sequence of characters");
```

Strings

- Strings can be initialised at the time of declaration using string literals

```
String str1 = new String("You should attend lectures");
```

a simplified form

```
String str2 = "Java is not hard to learn";
```

- You can use print(), println() or printf() methods to display strings

```
System.out.print(str1);
System.out.printf("%s \n %s \n", str2, str1);
```

- You can use **`Scanner`** class to input strings

```
Scanner userInput = new Scanner(System.in);
String fullName = userInput.nextLine(); // read a line
String firstName = userInput.next();    // read a word
```

Strings

- The + operator has another meaning for strings and can be used for their concatenation

```
String firstName = "Paul";
String lastName = "Deitel";
String author = firstName + " " + lastName;
```

- There are no `-`, `*`, or `/` operators defined for strings

Quiz

What is actually produced and sent to the monitor when you use print() method?

```
System.out.print("The distance is: " + distA + " km");
```

Strings

- There are several methods defined in the class String

- The method **`length()`** returns the number of characters stored in the string

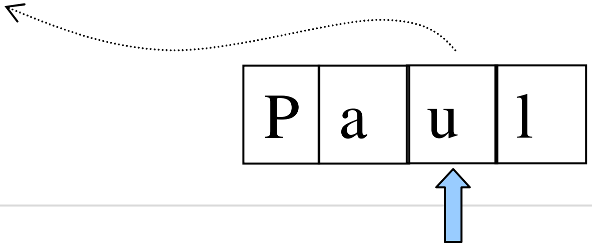
```
int strSize = author.length(); // get length
printf("%s contains %d characters\n", author, strSize);
```

- The method **`substring(start, size)`** obtains a sequence of **`size`** characters from a string starting at **`start`** position

```
String firstName = author.substring(0,4);  // Paul
```

- The method **`charAt(p)`** obtains a character from a string that corresponds to the position **`p`**

```
String name = "Paul";
char letter = name.charAt(2);  // u
```



Characters

- There are no methods defined in `Scanner` to read individual characters from the user input
- To read a character, you can:

1. use `next()` method to read a word, or `nextLine()`

```
Scanner keyboard = new Scanner(System.in);
String inWord = keyboard.next();  // read a word
```

2. Obtain the first character of the word

```
char letter = inWord.charAt(0);   // get first char
```

Note: Everything what you type before pressing *Enter* goes into the keyboard buffer first. Having read next word, the method **`next()`** removes it from the buffer. If you entered two words separated by whitespace and call **`next()`** method only once, the second word will remain in the keyboard buffer. It's safer to use **`nextLine()`** that will remove everything until **`cr`** from the buffer after reading.

Strings

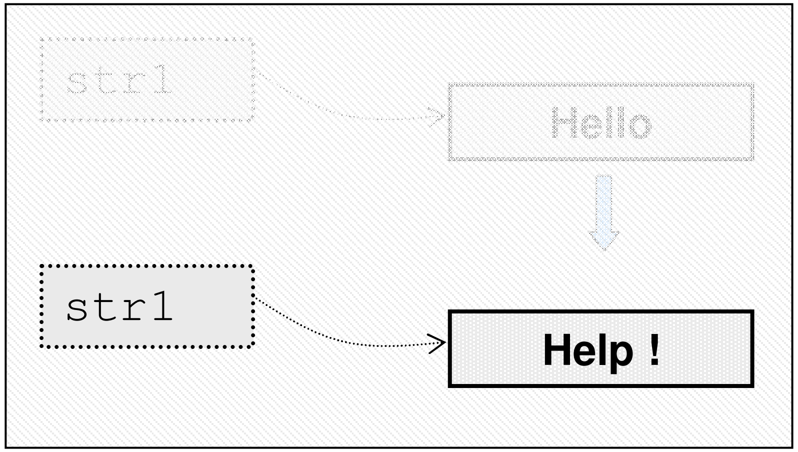
- The String class is immutable, that means once it is created a String object cannot be changed
- When it looks like an operation modifies a string, it actually creates a new one that contains the result of the operation

Example:

```
String str1 = "Hello";
str1 = str1.substring(0,3) + "p !";
print(str1);
```

```
Help !
```

A new reference variable with the same name is created that is linked to a new String object



Strings

- The majority of application do not modify strings, they compare strings
- String class has a method `equals` that compares two strings. It returns a `boolean` value `true` if they are equal, or `false` otherwise

```
import java.util.Scanner;
class Example
{
   public static void main(String[] args)  {
      String userName = "Peter";
      System.out.print("Enter your name: ");
      Scanner userInput = new Scanner(System.in);
      String typedName = userInput.nextLine();

      boolean theSame = userName.equals(typedName);
      System.out.print("Matched: " + theSame);
   }
}
```

```
java Example
Enter your name: Peter
Matched: true
```

Strings

- Strings which contain only symbols of digits may be converted to **`int`** or **`double`**

Example:

```
String strNumber = "158";
/* convert "158" to an integer value 158  */
int numberOfStudents = Integer.parseInt(strNumber);
```

```
class Example
{   /* read two command line arguments and find their maximum */
    public static void main(String[] args)  {
        String str1 = args[0];  // read 1st command line argument
        String str2 = args[1];  // read 2nd command line argument

        double num1 = Double.parseDouble(str1); // convert
        double num2 = Double.parseDouble(str2); // convert

        double result = Math.max(num1, num2);
        System.out.print("Max: " + result);
    }
}
```

```
java Example 1.2  5.7
Max: 5.7
```

Suggested reading

Java: How to Program (Early Objects), 11th Edition

- Chapter 2: Introduction to Java applications
- Appendix A
- Appendix B
- Appendix D