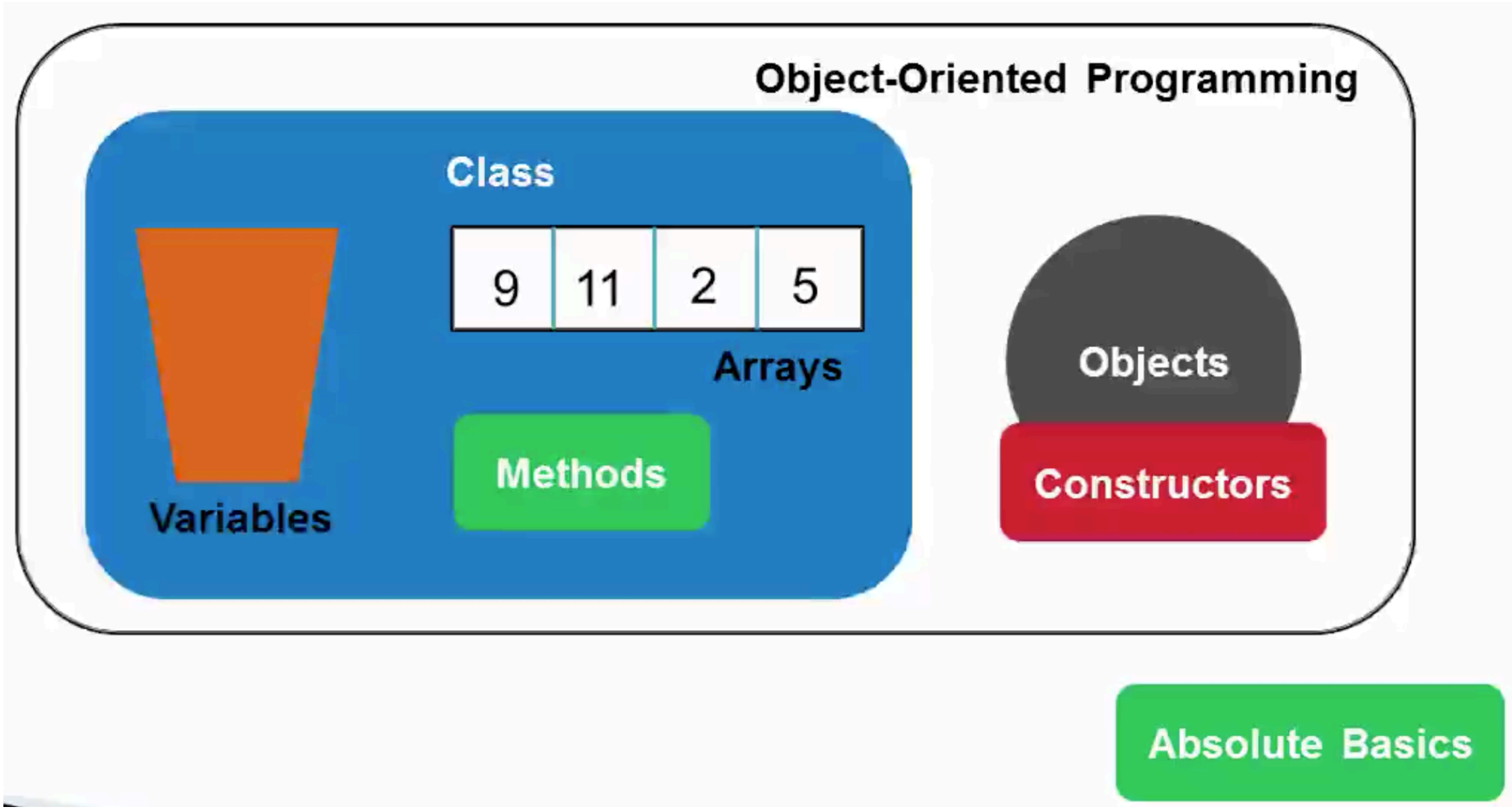
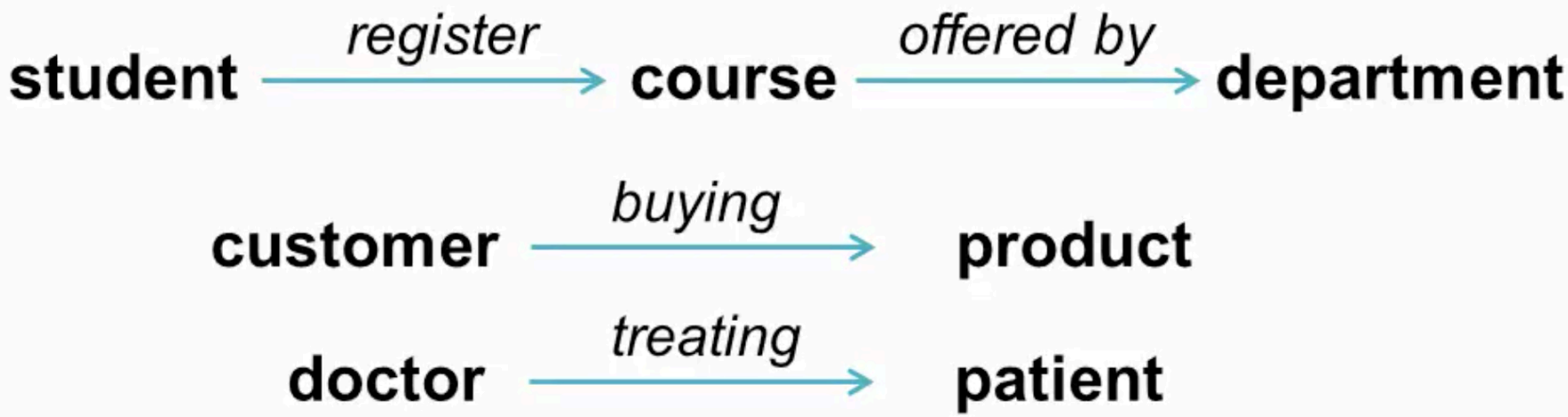
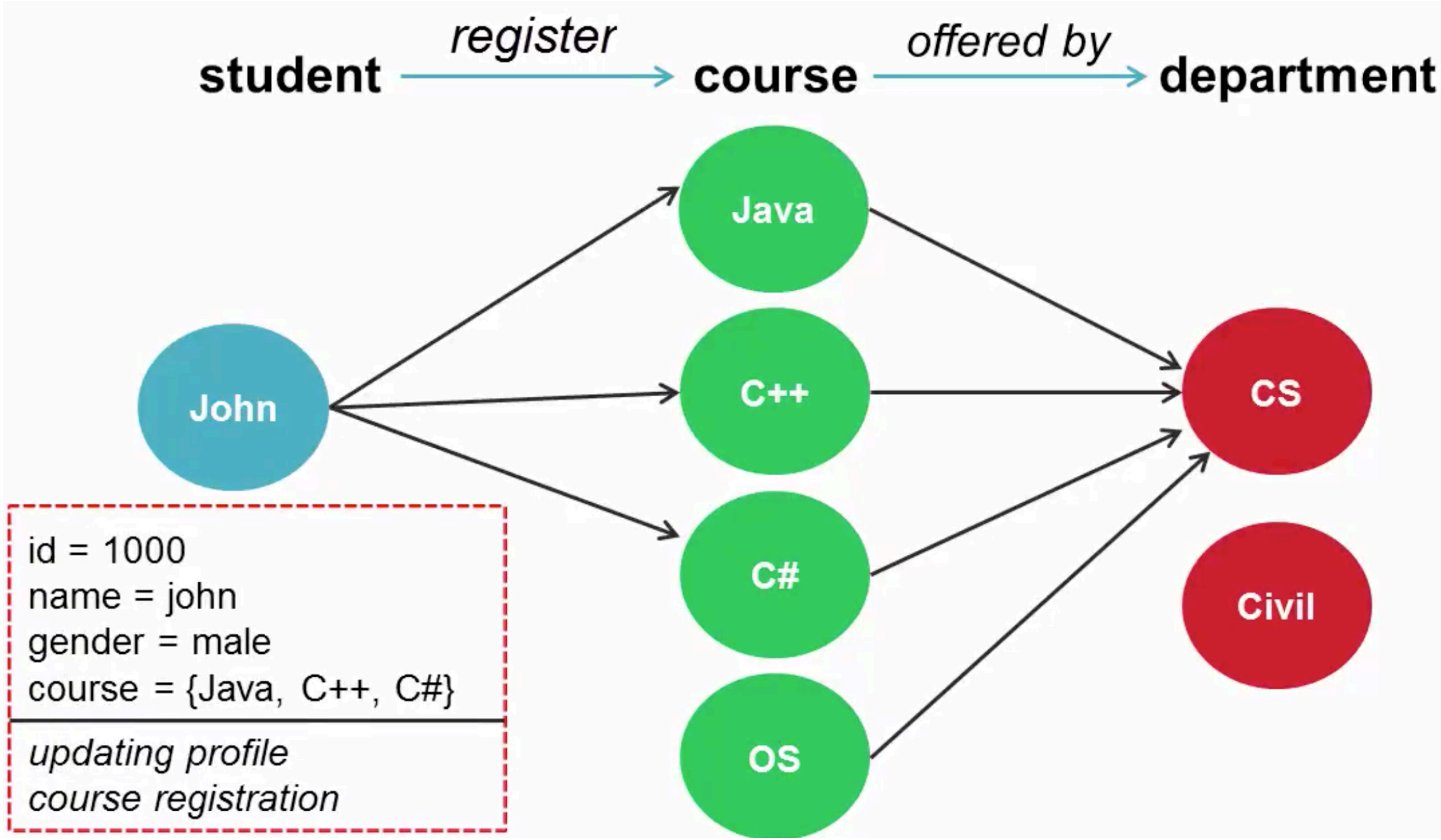
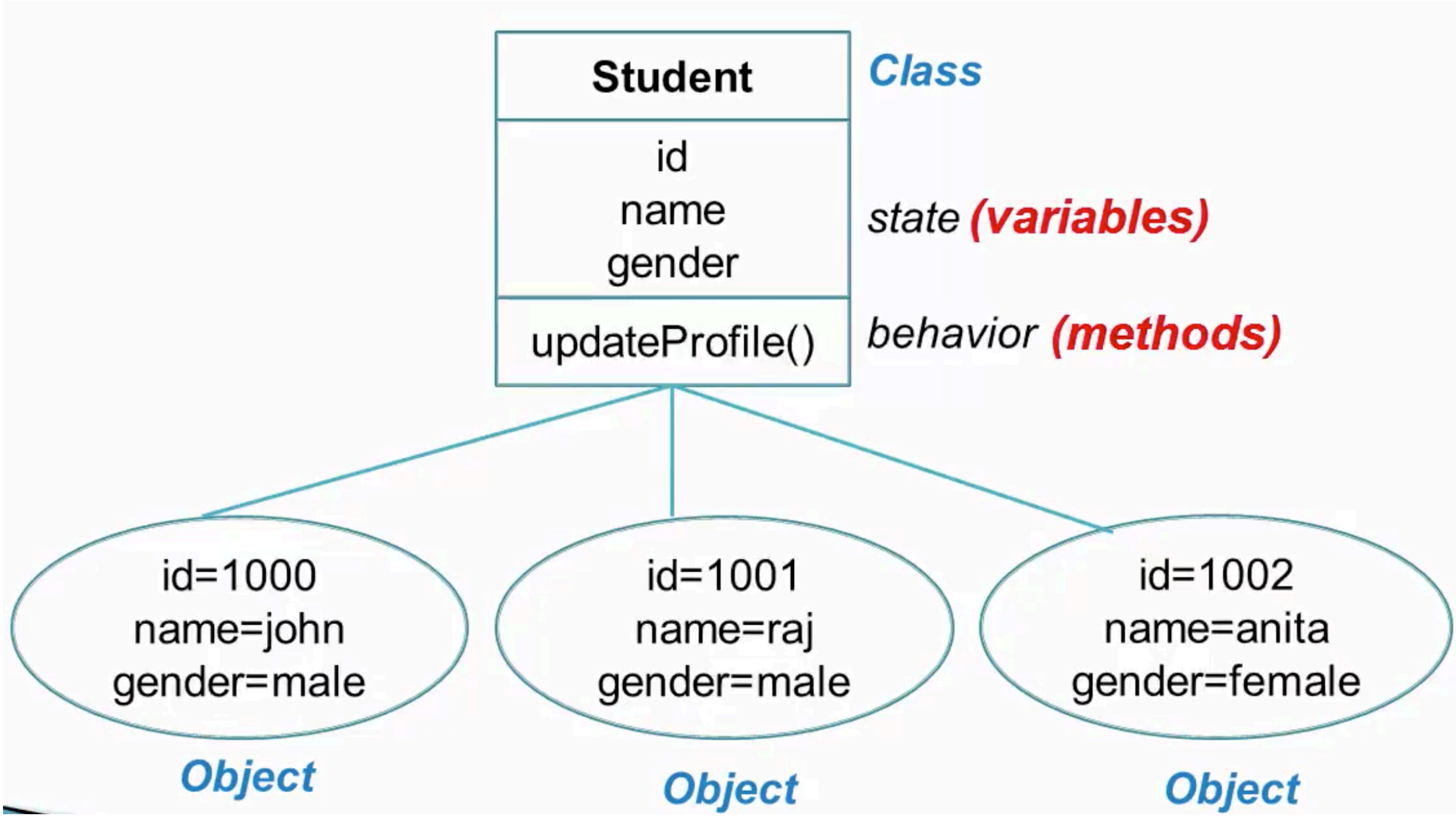

Object-Oriented Programming
Class
9
11
2
5
Arrays
Variables
Methods
Objects
Constructors
Absolute Basics

Object-Oriented Programming

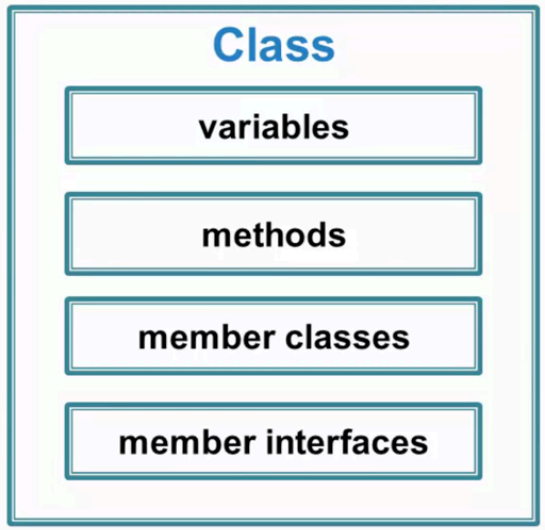
- Roots in 1960s
- Implement *large projects* in *simple* way
- Model ***real-world scenarios*** in a more *natural* way




student
register
course
offered by
department
Java
C++
C#
OS
John
CS
Civil
id = 1000
name = john
gender = male
course = {Java, C++, C#}
updating profile
course registration


Student
id
name
gender
updateProfile()
Class
state (variables)
behavior (methods)
id=1000
name=john
gender=male
id=1001
name=raj
gender=male
id=1002
name=anita
gender=female
Object
Object
Object

Class Members




```
class Student {
    // variable declarations
    int id;
    String name;
    String gender;

    // method definitions
    boolean updateProfile(String newName) {
        name = newName;
        return true;
    }
}
```

```
class StudentTest {
    public static void main (String[] args) {
        // 1. creating a new student object
        Student s = new Student();

        // 2. setting student's state
        s.id = 1000;
        s.name = "joan";
        s.gender = "male";

        // 3. updating profile with correct name
        s.updateProfile("john");
    }
}
```

Comments

- //
  - Double slash
  - Ignore rest of line
  - Single-line comments or disable single line of code
- /* */
  - Block quotes
  - Ignore everything between /* and */
  - Detailed code description or disable large areas of code

Case Sensitivity

Everything is **case-sensitive**

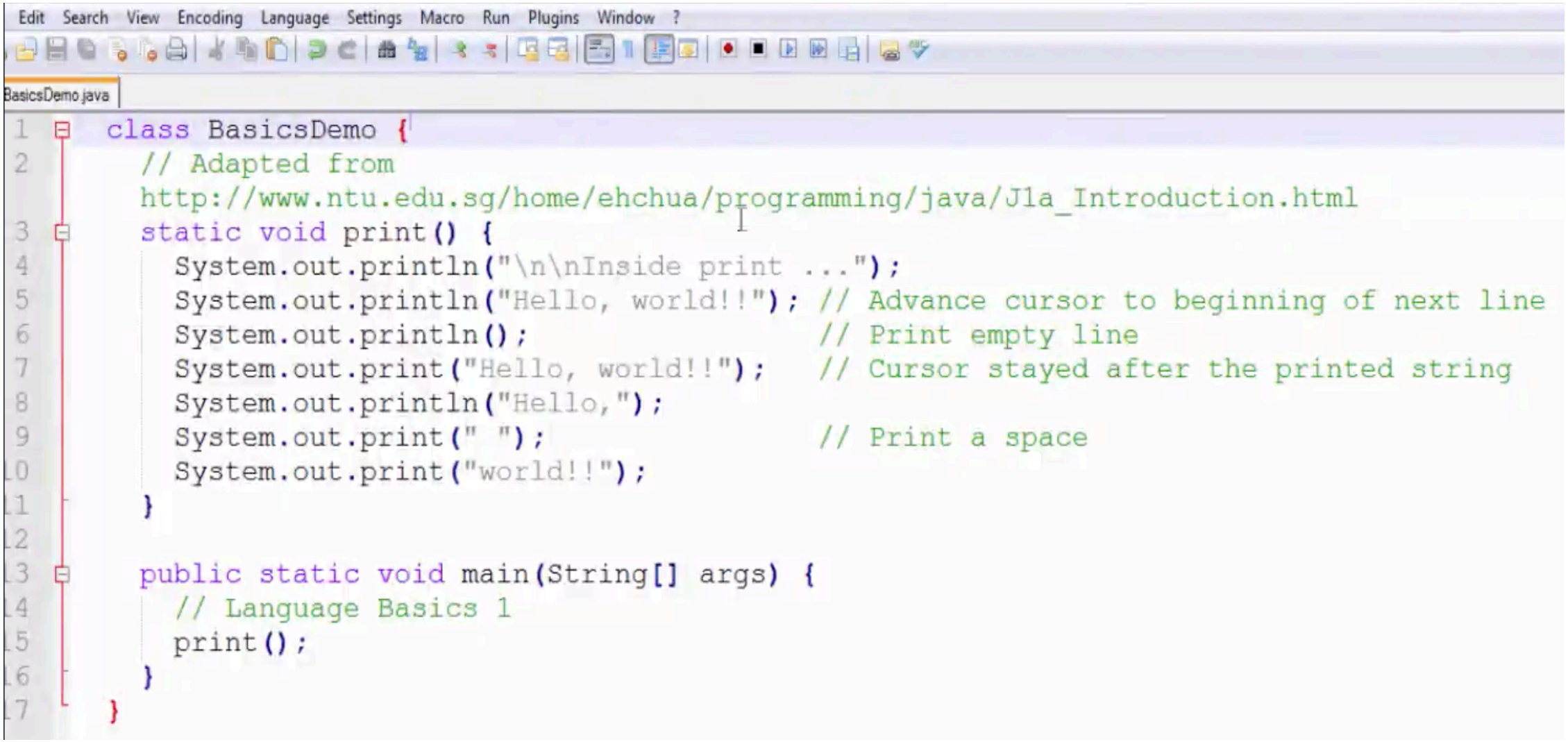
- Variable name *test* is different from *Test*

Naming Rules

- Classes, methods and variables
  - Must start with **letter**, **underscore**, or **$**
  - Other characters can be letters, underscore, $, or **numbers**
- No reserved keywords
  - class, interface, enum, abstract, implements, extends, this, super
  - byte, short, char, int, long, float, double, boolean
  - break, continue, for, do, while, if, else, new, switch, default
  - …

Printing to Console

- System.out.**println**(string)
  - Print and place cursor at *beginning* of next line
- System.out.**print**(string)
  - Print and place cursor *after* the printed string
- **Demo**: basics\BasicsDemo.print()


Edit Search View Encoding Language Settings Macro Run Plugins Window ?
BasicsDemo.java
class BasicsDemo {
// Adapted from
http://www.ntu.edu.sg/home/ehchua/programming/java/J1a_Introduction.html
static void print() {
System.out.println("\n\nInside print ...");
System.out.println("Hello, world!!"); // Advance cursor to beginning of next line
System.out.println(); // Print empty line
System.out.print("Hello, world!!"); // Cursor stayed after the printed string
System.out.println("Hello,");
System.out.print(" "); // Print a space
System.out.print("world!!");
}
public static void main(String[] args) {
// Language Basics 1
print();
}
}

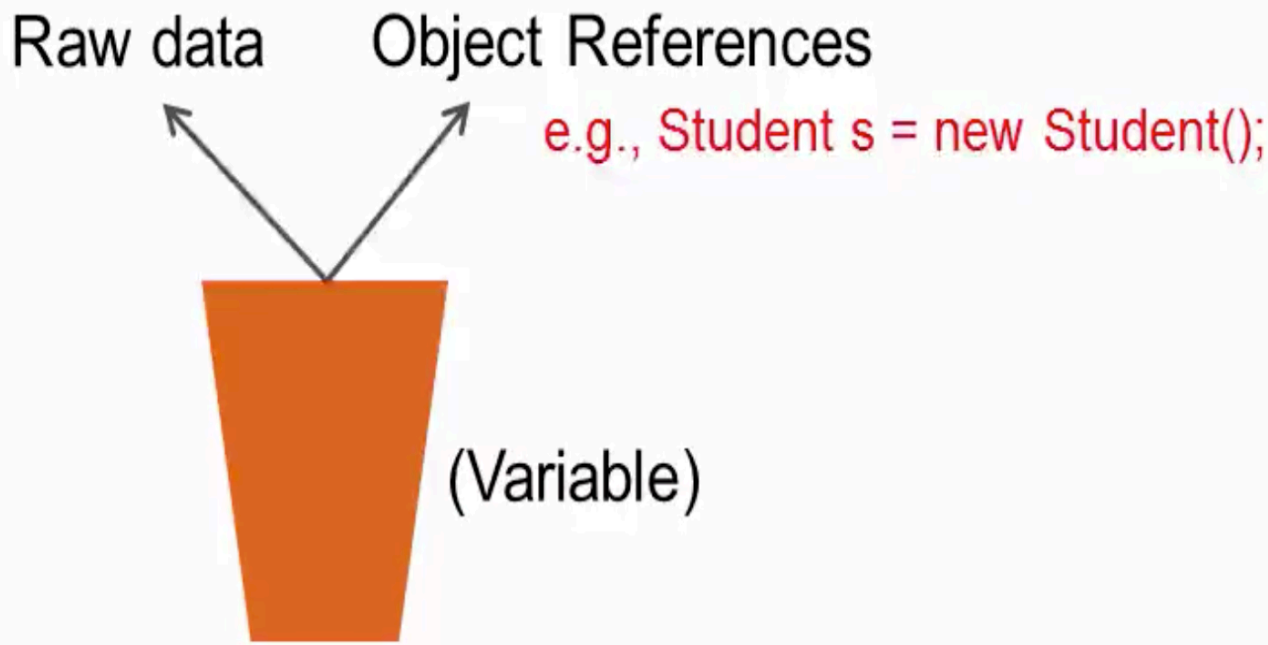
Java’s Reserved Keywords :

class, interface, enum, abstract, implements, extends, this, super
byte, short, char, int, long, float, double, boolean
break, continue, for, do, while, if, else, new, switch, default, case, goto
try, catch, final, assert, throw, throws
package, import
public, private, protected
synchronized, instanceof, return, transient, static, void, finally, strictfp, volatile, const, native

Following is the reference link from where the above information is taken:
http://docs.oracle.com/javase/tutorial/java/nutsandbolts/_keywords.html

Variable


1000
John
male


Raw data
Object References
e.g., Student s = new Student();
(Variable)

Variable Type

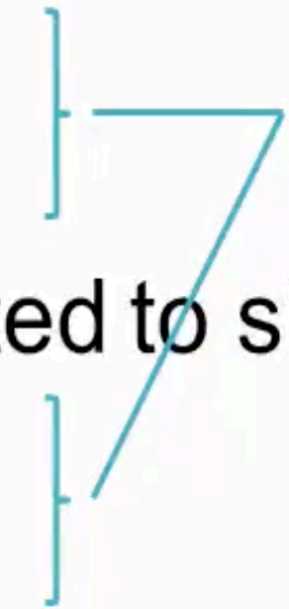
```
int  id = 1000;

String  name = “john”;

Student  s = new Student();  s.updateProfile(“john”);
```

Statically typed language → Static type checking

Dynamically typed language → Dynamic type checking

Variable Declaration

- <type> <name> [= *literal* or *expression*];
- Literal → raw data
  - int count = 25;
  - boolean flag = true;
- Expression → evaluated to single value
  - int count = x;
  - int count = getCount();
- Can appear *anywhere* in class



declaration statements

Reinitializing Variable

- Variable ~ *something whose value can be changed*
- **Assignment statement**
  - <variable-name> = literal or expression
  - count = 23;
  - count = x + y;

```
class Student {
    // variable declarations
    int id;
    String name;
    String gender;

    // method definitions
    boolean updateProfile(String newName) {
        name = newName;
        return true;
    }
}
```

Variable Kinds

- Instance variables
- Static variables
- Local variables

Instance & static variables ~ *fields* or *attributes*

Instance Variables

- Declared directly within class
- Represent **object state**
- Gets *default value*
- Cannot be *reinitialized* directly within class

Static Variables

- Declared directly within class with keyword **static**
- Class variables
  - One copy per class
- Gets *default value*
- Cannot be *reinitialized* directly within class

Local Variables

- Declared in methods
- Includes method parameters
- Not accessible outside method
- Don't get *default values*

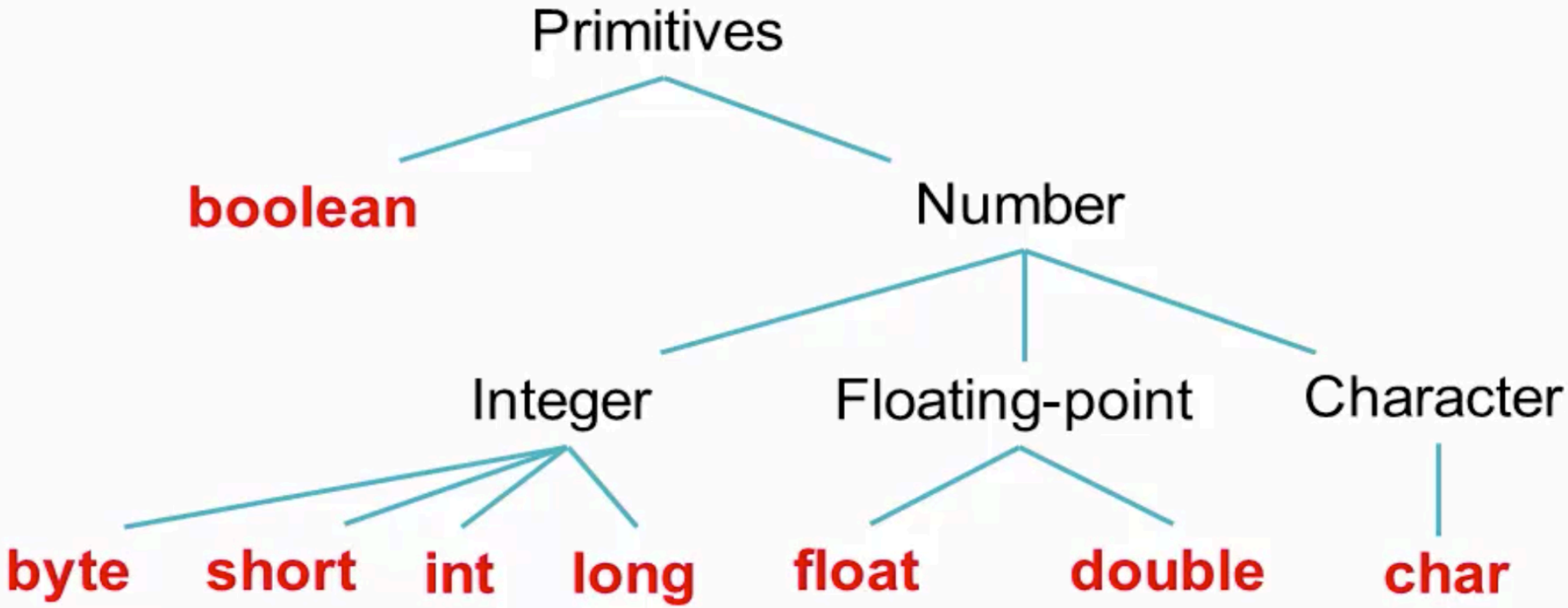
```
class Student {
    // instance variable
    int id;
    id = 22; // illegal
    static int count = 25; // static variable

    boolean updateProfile(String newName) {
        id = 22; // ok
        boolean result = true; // newName & result are local variables
        name = newName;
        return result;
    }
}
```

Variable Types

- Primitive
- Object references (non-primitive)

Primitive Types

8 primitives types



Integers

- Whole or fixed-point numbers, e.g., 65
- *byte*, *short*, *int*, *long*
- Data representation ~ **Signed two's complement**

Type	Bit depth	Value range	Default
byte	8 bits	-2^7 to $2^7 - 1$	0
short	16 bits	-2^{15} to $2^{15} - 1$	0
int	32 bits	-2^{31} to $2^{31} - 1$	0
long	64 bits	-2^{63} to $2^{63} - 1$	0

Integer Examples

- byte b = 100;
- short s = 1000;
- int i = -10000;
- long l = 1000000L; // L required if value above int range

Other Integer Literal Formats

- int y = **0x**41; // hexadecimal. y = 65 ($4 * 16^1 + 1 * 16^0$)
- int y = **0b**01000001; // (Java 7) binary
- int y = **0**101; // octal

Integer Examples

- int x = 5, y = 6;
- int x, y = 6;
  - Class-level: x = 0
  - Methods: x needs to be initialized before use
- int x = y = 99; // y is first **reassigned** to 99

Below is the reference link to an excellent article on Data Representation. It includes very clear explanations on topics like signed two's complement, IEEE 754 floating point scheme, and character encoding. I would say it is a must read if you are not familiar with these topics.

http://www3.ntu.edu.sg/home/ehchua/programming/java/DataRepresentation.html

Below is also another link that discusses some basic computer science stuff. One of the links there talks about signed two's complement scheme. If needed, you can take a look at it too.

https://users.cs.duke.edu/~raw/cps104/TWFNotes/html/twoscomp.html

Floating-point Numbers

- Real numbers, e.g., 3.14
- 32-bit **float** or 64-bit **double**
  - double is more precise
- Data representation ~ 32 & 64-bit **IEEE 754 floating point**

Type	Bit depth	Value range	Default
float	32 bits	-3.4E38 to 3.4E38	0.0f
double	64 bits	-1.7E308 to 1.7E308	0.0d

Floating-point Examples

- float f = 123.4f; // MUST end in f or F
- double d = 123.4; // trailing d or D is optional
- float e = 1.39e-23f; // $1.39 * 10^{-23}$

Rules of Thumb

- **Item 48: Avoid float and double if exact answers are required**
  - float & double ~ ill-suited for *monetary* calculations
  - Use **BigDecimal**
  - int or long ~ *keep track of decimal point yourself!*
- Floating-point is not as fast as integer arithmetic
- Stick with **int** & **double**
  - double's preciseness ~ useful in *neural networks*
- Use byte, short, float only if *memory saving* is important

Characters

- Single letter characters, e.g., ‘a’, ‘A’, ‘0’
- 16-bit ***char***
- Data representation ~ **16-bit unsigned integer**

Type	Bit depth	Value range	Default
char	16 bits	0 to $2^{16} - 1$	‘\u0000’

- \u → Unicode escape sequence

Characters

- Java uses 16-bit **Unicode** (UTF-16) scheme
  - Unicode → represents all characters in all languages
  - UTF-16 encoding scheme
  - ‘A’ → 0041 → **00000000 01000001**
  - Checkout Unicode table at **http://unicode-table.com**

Character Examples

- char c = ‘A’; // single quotes
- char c = 65; // prints A. In Unicode, 65 → A
- char c = ‘\u0041’; // $4 * 16^1 + 1 * 16^0$ → 65 → ‘A’ 
- char c = 0x41; // $4 * 16^1 + 1 * 16^0$ → 65 → ‘A’
- char c = 0b01000001; // binary → 65 → ‘A’
- Checkout reference on data representation
- **Demo**: basics\BasicsDemo.*primitives*()

Boolean

- boolean
  - Binary type ~ true/false

Type	Bit depth	Value range	Default
boolean	JVM specific	true or false	false

- boolean result = true;

Type	Bit depth	Value range	Default
byte	8 bits	-2^7 to $2^7 - 1$	0
short	16 bits	-2^{15} to $2^{15} - 1$	0
int	32 bits	-2^{31} to $2^{31} - 1$	0
long	64 bits	-2^{63} to $2^{63} - 1$	0
float	32 bits	-3.4E+38 to 3.4E+38	0.0f
double	64 bits	-1.7E+308 to 1.7E+308	0.0d
char	16 bits	0 to 2^{16} - 1	‘\u0000’
boolean	JVM specific	true or false	false

BigDecimal should be used when it comes to monetary calculations. When it comes to monetary calculations, BigDecimal is preferred for its precision & rounding strategies.
Mostly double should be good for precision concerns, but rounding strategies that BigDecimal class offers out-ofthe-box seems to be its big advantage. This is very useful for monetary calculations.
Take a look at the following article that explains BigDecimal from monetary standpoint. It explains how one rounding strategy (ROUND_CEILING) is useful when calculating tax while another while charging the cost external to tax (ROUND_HALF_UP). When using double, programmer has to apply such rounding strategies manually and that could be clumsy and errors can creep in.

http://www.opentaps.org/docs/index.php/How_to_Use_Java_BigDecimal:_A_Tutorial

http://www.javapractices.com/topic/TopicAction.do?Id=13

Type Casting

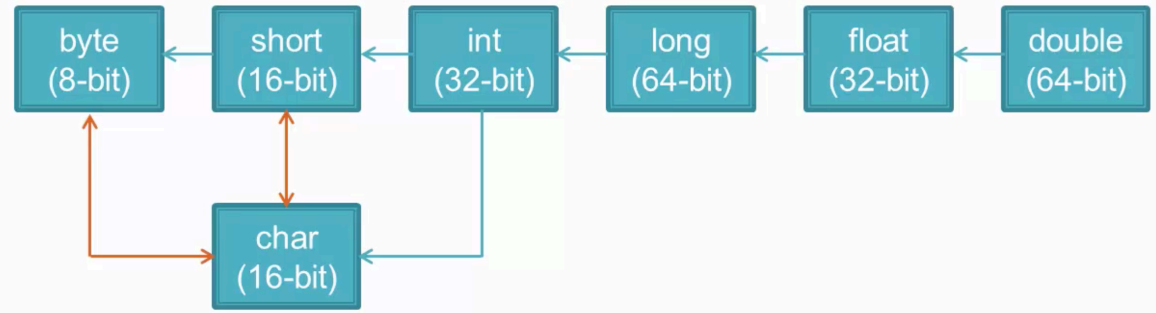
Assign *variable* or *literal* of one type to *variable* of another type

- int ← long
- int ← byte

- Only ***numeric-to-numeric*** casting is possible
- Cannot cast to *boolean* or vice versa

Explicit Casting

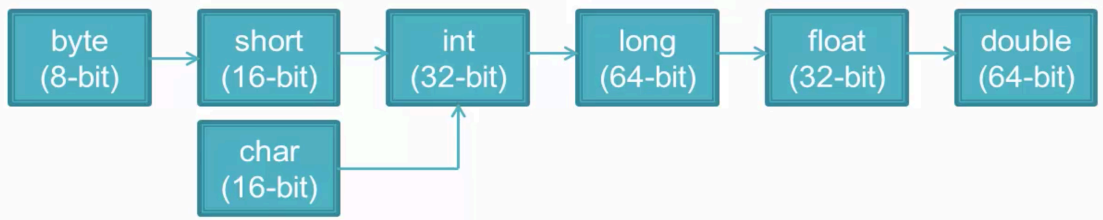
Larger to smaller ~ *narrowing* conversion



Implicit Casting

- Smaller to larger ~ *widening* conversion

  int x = 65;
  long y = x; *(Implicit casting by compiler)*



- Integer to Floating-point is *implicit* too

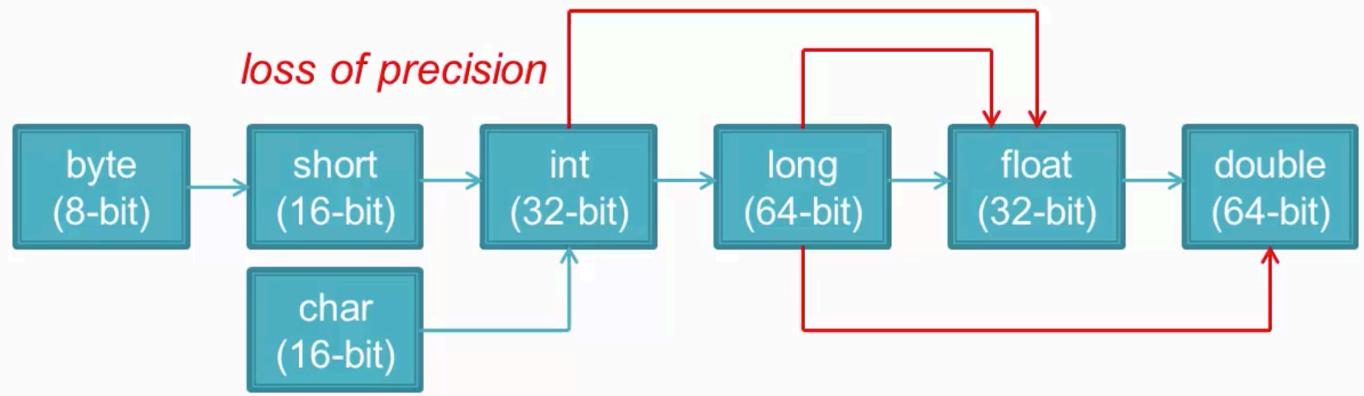
Explicit Casting Examples

- long y = 42;
  int x = **(int)** y;
- byte b = 65;
  char c = **(char)** b; // c = **'A'** *(widening & narrowing)*
- char c = 65; // c = '**A**'
- short s = 'A'; // s = **65**

Information Loss in Explicit Casting

- **Out-of-range** assignments
  - byte narrowedByte = (byte) 123456; // ***64***
- **Truncation**
  - Floating-point to integer/char will always truncate
  - int x = (int) 3.14f; // x = **3**
  - int y = (int) 0.9; // y = **0**
  - char c = (char) 65.5; // c = 'A'

Information Loss Implicit Casting




```
int oldVal = 1234567890;
float f = oldVal; // implicit cast
int newVal = (int) f; // 1234567936
```

Casting Use-cases

- Implicit Casting

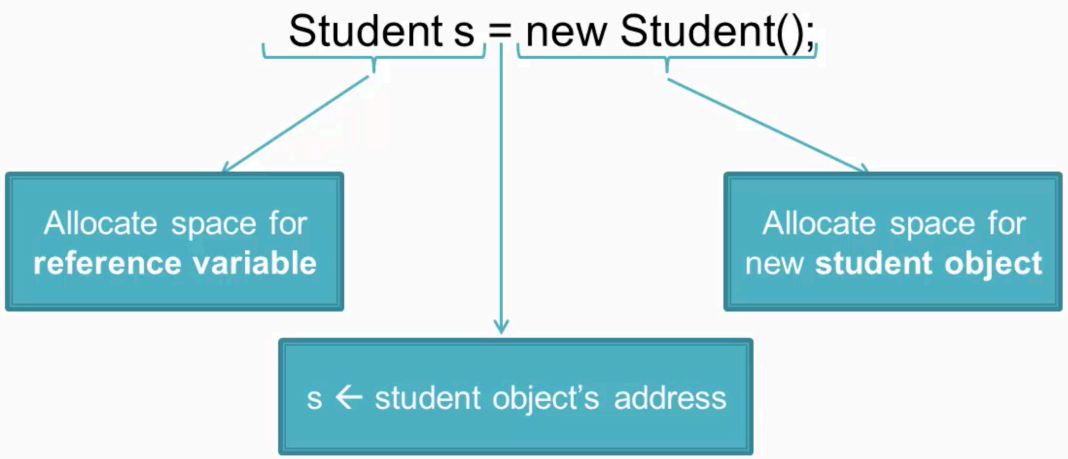
```
float f1 = 3.133f;
float f2 = 4.135f;
go(f1, f2);
```

```
go(double d1, double d2) {
    …
}
```

- Explicit casting

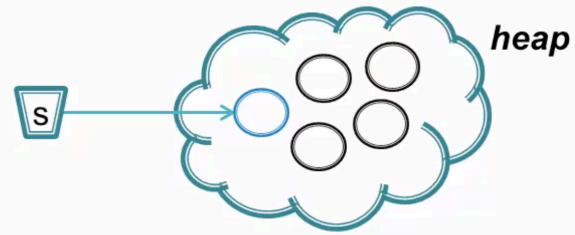
```
double avg = (2+3)/2; // 2.0, not 2.5
double avg = (double) (2+3)/2;
```

Object References



Where are Objects Stored?

Objects live on *heap*



Bit Depth & Default

- Bit depth ~ JVM specific
- Default ~ **null**
  - Student s; // s is *null* until initialized
    s.updateProfile(); // **NullPointerException**

Statements

- ***Command*** to be executed
  - Declare a variable
  - Change variable value
  - Invoke a method
- Change program state

Statement Types

- Declaration statements, e.g., int **count = 25**;
- Expression statements
  - count = 25; *// assignment statement*
  - getCount(); *// method invocation statement*
  - count++; *// increment statement*
- Control flow statements

not at class-level

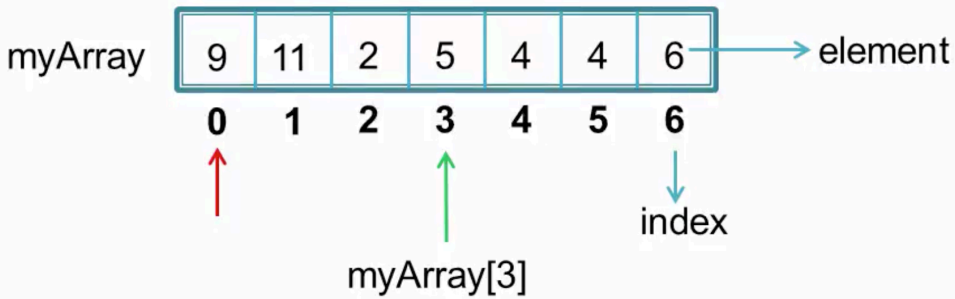
```
if (count < 100) {
    …
}
```

- Involves one or more ***expressions***
- Expression ~ evaluated to **single value**
  - Involves literals, variables, operators, and method calls
- Example
  - int count = x * getCount();
  - x, getCount(), x * getCount(), count = x * getCount()

compound expressions

Arrays

Container ***object*** that holds **fixed** # values of **single type**



Creating an Array

- int[] myArray = **new** int**[**7**]**;
  - Each element gets *default* 0
- myArray[0] = 9;
- myArray[1] = 11;
- myArray[2] = 2;
- myArray[3] = 5;
- myArray[4] = 4;
- myArray[5] = 4;
- myArray[6] = 6;

- int[] myArray = new int[]**{**9, 11, 2, 5, 4, 4, 6**}**;
- int[] myArray = {9, 11, 2, 5, 4, 4, 6};

int myArray**[]**; // ok

Arrays

- **length**
  - myArray.length → 7
- Accessing outside array boundary → runtime error
  - int item = myArray[7]; // *runtime error*

Array of Object References

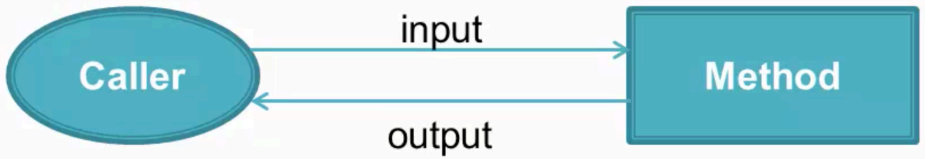
- **Student**[] students = new **Student**[2];
  - students[0] & students[1] → *null*
- students[0] = new Student();
  students[1] = new Student();
- students[0].name = “john”;
  students[1].name = “raj”;

Random Access

- Linear layout → *fast random access*, **O(1)**
- myArray[1000] ~ myArray[5]
- Searching ~ **O(n)**

Methods

- Define *behavior*
- **Self-contained logic** that can be used *many* times
- Can receive *input* & generate *output*



Syntax

signature *parameters or formal parameters*

```
returnType methodName(type param1, type param2,... ) {
    …
    return someValue;
}

type var =  methodName(arg1, arg2, …)
```

arguments or actual parameters

Return Type

- **void**
  - Nothing to return
  - Optional **return;** as last statement
  - void print() {
    System.out.println("Hello World!!");
    }
- Must be *primitive*, *object reference*, or *void*
- Primitive or object reference → *must* return value

Examples – Method Definition

- int getId() {
  byte x = 5;
  return x; // *compatible* with int
  }
- byte getId() {
  int x = 5;
  return **(byte)**x;
  }

Examples – Method Invocation

- int **search**(int[] list, int key) {
  …
  }
- search(**new int[]**{5, 13, 2, 1}, 1);
- search({5, 13, 2, 1}, 1); // compiler error

Method Benefits

- Avoid duplicate code
- Divide and conquer
  - Software reuse
  - *Clean* and *readable* code

```
int search (int[] list, int key) {
  // Step 1: Sort
  …
  …
  …

  // Step 2: Binary search
  …
  …
  …
  …
}
```

```
int search (int[] list, int key) {
  sort(list);
  binarySearch(list, key);
}
```

Instance Methods

- Object-level methods
- **Invocation**: *objectRef.methodName()*
- Affect *object state*
  - Instance variables
  - Other instance methods

Static Methods

- Keyword **static** in declaration
- Class-level methods
  - No access to *state*, i.e., can't access instance variables/methods
- Can access *static variables*
- **Invocation**: *className.methodName()*
- *main* method is static
- Can access other static methods

```
void updateId(int newId) {
    newId = 1001;
}

int id = 1000;
updateId(id);
```

id = 1000 or 1001?

```
void updateId(Student s1) {
    s1.id = 1001;
}

Student s = new Student();
s.id = 1000;
updateId(s);
```

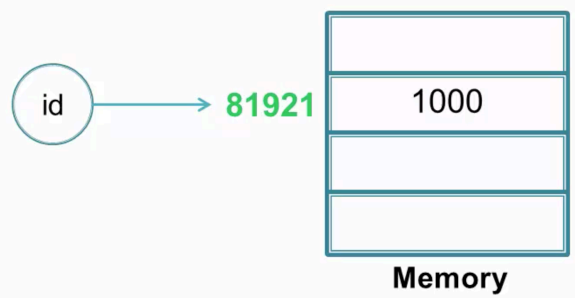
s.id = 1000 or 1001?

Passing Data

- Pass by value 
- Pass by reference

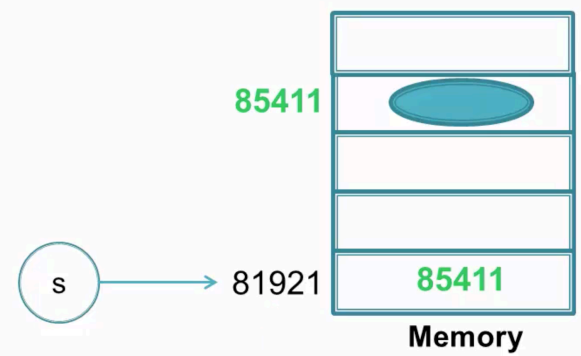
Primitives in Memory

- int id = 1000;
- id → <logical name, memory address, value>



Object References in Memory

- Student **s** = new Student();



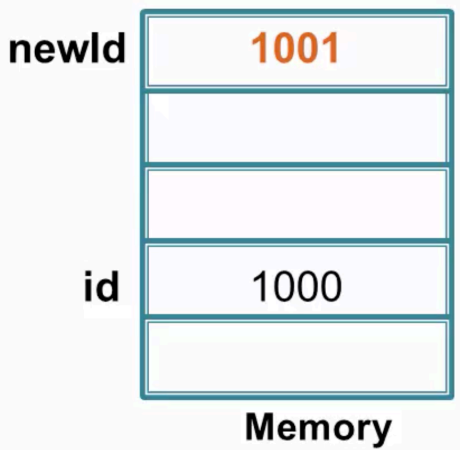
Pass by Value

Value of argument is passed to parameter

- Primitive argument ~ value is *primitive*
- Object reference argument ~ value is *memory address*

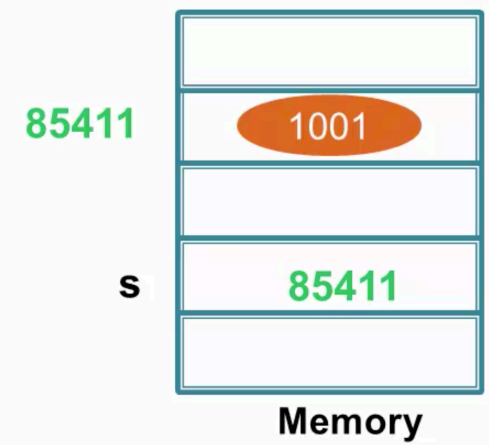
Pass by Value: Primitives

```
void updateId(int newId) {
      #2
   newId = 1001;
      #3
}

int id = 1000;
      #1
updateId(id);
      #4
```



Pass by Value: Object References

```
void updateId(Student s1) {
      #2
   s1.id = 1001;
      #3
}

Student s = new Student();
s.id = 1000;
      #1
updateId(s);
      #4
```



```
void updateId(int newId) {
    newId = 1001;
}

int id = 1000;
updateId(id);
```

id = 1000 or 1001?

```
void updateId(Student s1) {
    s1.id = 1001;
}

Student s = new Student();
s.id = 1000;
updateId(s);
```

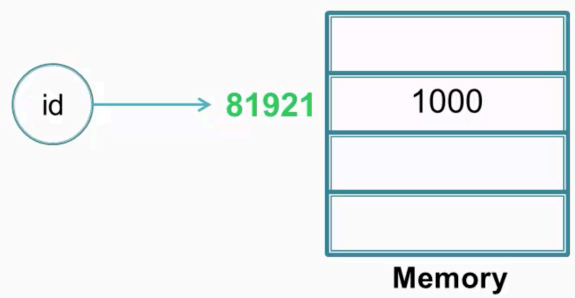
s.id = 1000 or 1001?

Passing Data

- Pass by value 
- Pass by reference

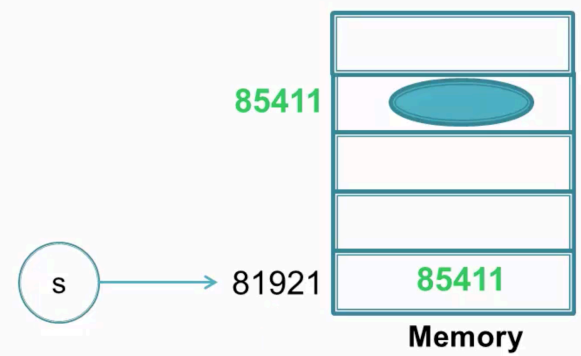
Primitives in Memory

- int id = 1000;
- id → <logical name, memory address, value>



Object References in Memory

- Student **s** = new Student();



Pass by Value: Reassignment

```
void updateId(Student s1) {
      #2 ✓
   s1 = new Student();
   s1.id = 1001;
      #3 ✓
}

Student s = new Student();
s.id = 1000;
      #1 ✓
updateId(s);
      #4
```

Passing Data

Identical to *variable assignment*

method parameter = method argument

```
int id = 1000;
int newId = id;
newId = 1001;
```

```
Student s = new Student();
s.id = 1000;
Student s1 = s;
s1.id = 1001;
```

Java is always pass by value!!

Method Overloading

- *Same name*, *different parameter lists*
- ***MUST*** change parameter list
  - # parameters or parameter types or both must vary
- Changing only *return type* doesn't matter
- Applies to *instance* & *static* methods

Invalid Examples

```
void updateProfile(int newId) {}
```

- *boolean* updateProfile(int newId) {}
- void updateProfile(int *id*) {}
- *static* void updateProfile(int id) {}

Valid Examples

- boolean updateProfile(int newId) {}
  boolean updateProfile(int newId, char gender) {}
  boolean updateProfile(char gender, int newId) {}
- void overload(**int** i) {}
  void overload(**byte** b) {} **void overload(short s) {}**
- overload(23);
- byte b = 23;
  overload(b);

varargs

- Before Java 5 ~ *fixed* # arguments
- *varargs* ~ variable-length arguments
- **Last** parameter can take ***variable*** # arguments
  - Can be the only parameter

Syntax & Invocation

- **Syntax**: three dots following parameter type
  - foo(boolean flag, **int…** items)
- Invocation
  - *Array*: foo(true, **new int[]{1, 2, 3}**)
  - *Comma-separated arguments*: foo(true, **1, 2, 3**)
  - *Omitted*: foo(true)

infinitely overloaded

varargs Restrictions

- Must be *last* parameter
  - foo(**int…** items, boolean flag) - **illegal**
- Only one varargs parameter
  - foo(boolean flag, **int…** v1, **int…** v2) - **illegal**

Why varargs?

- Can't we use *foo*(boolean flag, **int[]** items)?
- varargs provides ***simpler*** & ***flexible*** invocation
  - foo(true, **1, 2, 3**)
  - foo(true), i.e., no *foo*(true, null) or *foo*(true, new int[]{})
  - foo(true, veryLargeArray)
- **printf**(String format, Object… args)
  - System.out.printf("DOB: %d/%d/%d", 1, 1, 1978); //DOB: 1/1/1978

varargs & main Method

public static void main(**String[]** args) {}

or

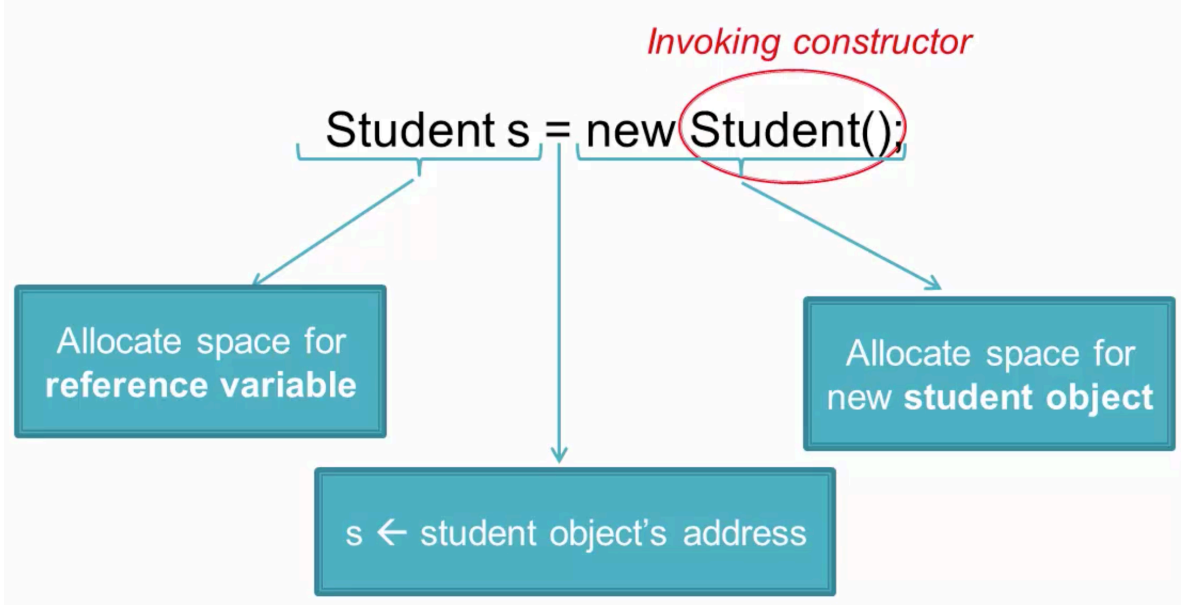
public static void main(**String…** args)

varargs & Overloaded Methods

- Invalid overload example
  - foo(boolean flag, int… items)
  - foo(boolean flag, **int[]** items)
- *varargs* method will be matched last
- **Demo**: basics\BasicsDemo.varargsOverload()

Constructor

- Runs on *object creation*
- Initializes ***object state***



Syntax

```
ClassName(type param1, type param2, ... ) {
        …
}
```

Example

```
class Student {
    int id;
    Student(int newId) {
        id = newId;
    }
}

Student s = new Student(1001);
```

Default Constructor

Constructor not provided

↓

Compiler ***inserts*** one with no parameters

no-args constructor

Example

```
class Student {
    int id;
}
```

→

```
class Student {
    int id;
    Student() { }
}
```

```
Student s = new Student();
```

Example

```
class Student {
    int id;
    Student(int newId) {
        id = newId;
    }
}
Student s = new Student(1001);
Student s = new Student(); // illegal
```

Constructor Overloading

- Same *overloading rules* as methods
  - Parameter list must be different
- Can create objects using *any* of the constructors

Why constructor overloading?

✓ To create *objects with different capabilities*

Example 1

FileOutputStream(String name, boolean append)

FileOutputStream(String name)

FileOutputStream(File file)

FileOutputStream(File file, boolean append)

FileOutputStream(FileDescriptor fdObj)

```
class User {
    int id;
    String name;
    int salary;

    User(int userId, String userName) {
        id = userId;
        name = userName;
    }
    User(int userId, String userName, int userSalary) {
        this(userId, userName);
        salary = userSalary;
    }
}
```

- ✓ MUST be first statement
- ✓ Only one-per-constructor
- ✓ Cannot use instance variables as arguments
- ✓ No recursive invocation

Recursive Constructor Invocation 2

```
class User {
    int id;
    String name;
    int salary;

    User(int userId, String userName) {
        this(userId, userName, 0); // compiler error
    }

    User(int userId, String userName, int userSalary) {
        this(userId, userName); // compiler error
    }
}
```

return Statement

```
class Student {
  Student(int id) {
    …
    return;
    // unreachable code
  }
}
```

this Reference

- *objectRef*.someVariable or *objectRef*.someMethod()
- *this*.someVariable or *this*.someMethod()
- *Reference to* ***current object***
- Members can be accessed *directly* too

So, why ***this*** *reference??*

Access instance variables ***hidden*** (or shadowed) by local variables

Example

```
class Student {
    int id;
    String name;

    void updateProfile(int newId, String name) {
        id = newId;
        this.name = name;
    }
}
```

Cannot be used in *static* methods