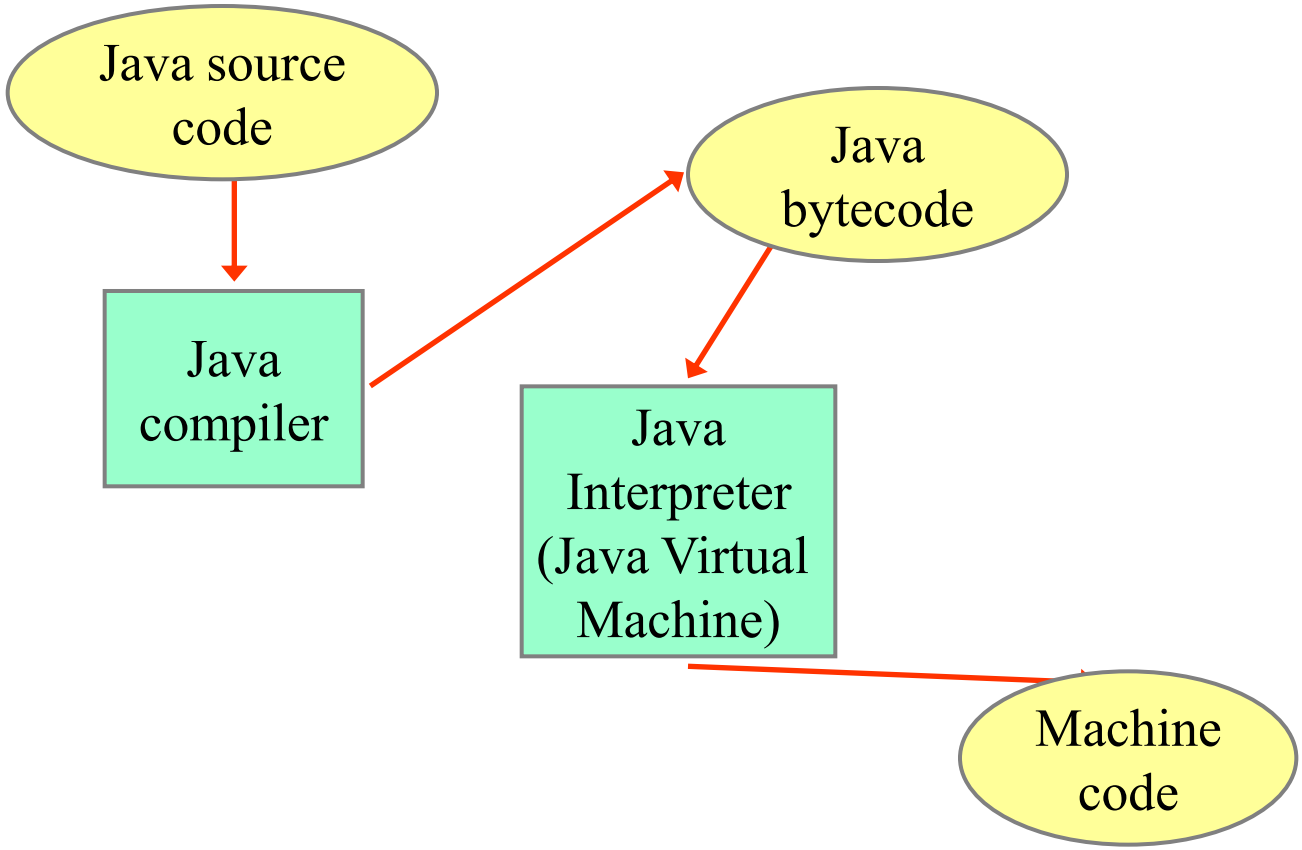
ITI 1121
Introduction to Computing II

Lecture 02-
2016

These slides are based on :

ITI1120/ITI1121 slides by Profs: D. Inkpen/M. Turcotte

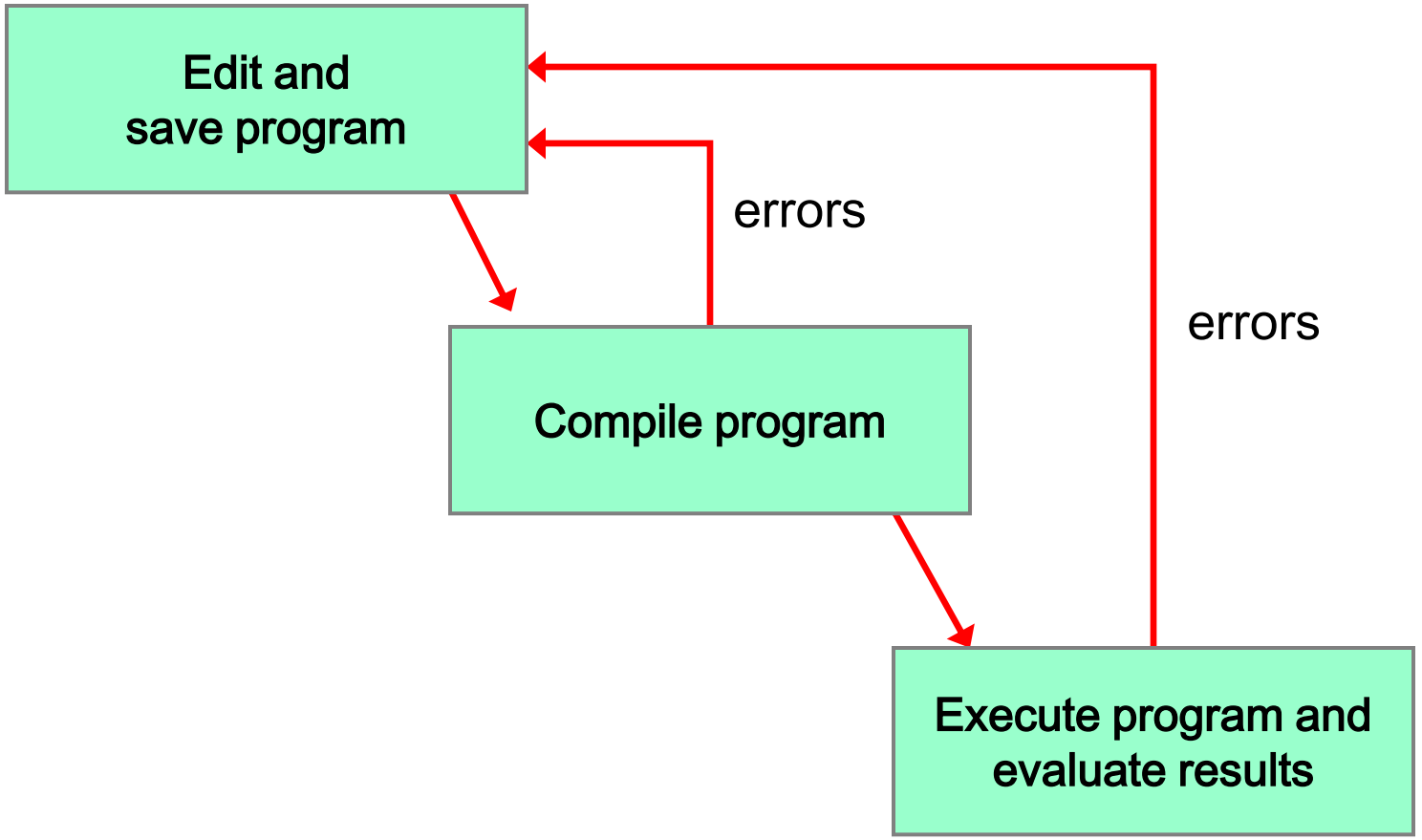
Remember: Translation in Java



Agenda

1. **Java programs**
2. What is a variable?
3. What is a data types?
    1. Primitive data types.
    2. Reference data types
        1. Example 1:Arrays
        2. Example 2: Strings
4. Methods in Java
5. Classes in Java

Java programs: Basic Program Development



Java programs: Three Types of Errors

1. Syntax errors: These are illegal combinations of symbols that do not obey the rules of the programming language.
   - Symptom: the program will not compile.

2. Run-time errors: These are errors which result from the data values used.
   - Symptom: the program crashes while running.

3. Logic (semantic) errors: These are errors which result from incorrect reasoning in the program. They probably occur because the algorithm is wrong.
   - Symptom: the program runs, but the results are wrong.

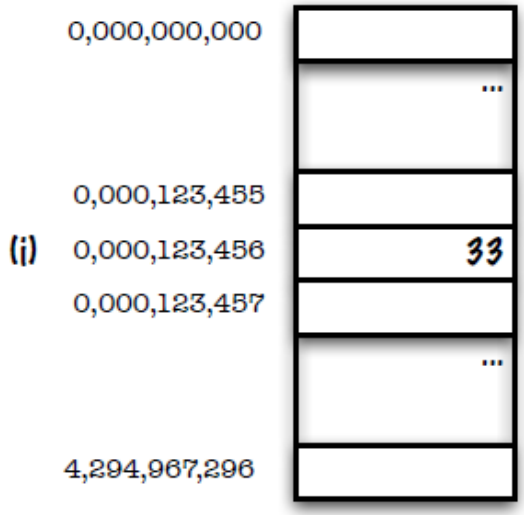
Agenda

1. Java programs:
2. **What is a variable?**
3. What is a data types?
    1. Primitive data types.
    2. Reference data types
        1. Example 1:Arrays
        2. Example 2: Strings
4. Methods in Java
5. Classes in Java

1.What is a variable

What is a variable?

- A variable is a place in memory, to hold a **value**, which we refer to with help of a **label**




```
byte i = 33;
```

1. What is a variable?

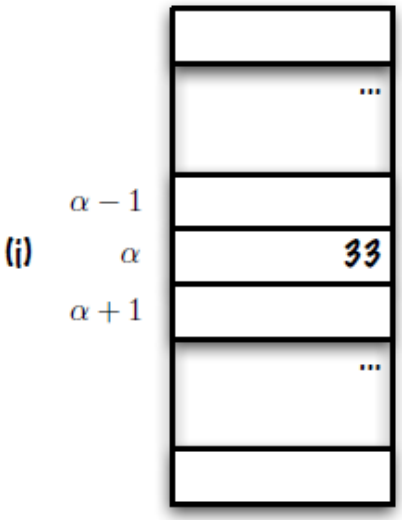
- To use a variable, you must declare it first.
  - A declaration of a variable specifies its type, its name, and, optionally, its initial value.
  - A declaration reserves memory for the variable.
  - Examples

```
int x = 0;                  // An int variable x with initial value 0
double d;                   // A double variable not initialized
char c = ' ';               // A char variable initialized as a blank
boolean b1 = false;         // A boolean variable initialized to false
```

- Variable declarations end with a semicolon.

1.What is a variable

I will be using Greek letters to designate memory locations (addresses) since in Java we don't know the location of "objects" and should not care!




```
byte i = 33;
```

Agenda

1. Java programs:
2. What is a variable?
3. **What is a data types?**
    1. Primitive data types.
    2. Reference data types
        1. Example 1:Arrays
        2. Example 2: Strings
4. Methods in Java
5. Classes in Java

2.What is a Data Type?

What are data types for?

- Yes, it tells the compiler how much memory to allocate:

```
double formula;          // 8 bytes
char c;                  // 2 bytes
```

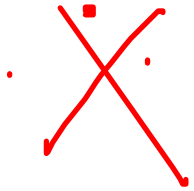
- But it also? It gives information about the meaning (semantic) of the data: **which operations are allowed**, which data are compatible. Hence the following statement,

```
c = flag * formula;
```

will produce an error at compile time; data types are therefore also useful to help detect errors in programs early on.

2.What is a Data Type?

- To be more precise, there are **concrete data types** and **abstract data types**
- Concrete data types specify both, the allowed operations **and** the representation of the data
- **Abstract Data Types (ADTs) specify only the allowed operations**



2.What is a Data Type?

In Java, we have primitive and reference data types:

- Primitive are:
  - numbers, characters (but not Strings) and booleans
  - **the value is stored at the memory location designated by the label of the variable**
- References:
  - Predefined:
    - Arrays
    - Strings
  - User defined, reference to an instance of a class;

Primitive data types

- A data item has a PRIMITIVE type if it represents a single value, which cannot be decomposed.

- Java has a number of pre-defined primitive data types. Examples are:

`int`	represents integers (e.g. `3`)
`double`	represents "real" numbers (e.g. `3.0`)
`char`	represents single characters (e.g. `'A'`)
`boolean`	represents Boolean values (e.g. `true`)

An example: The Type `int`

- A variable of type `int` may take values from `-2147483648` to `2147483647`.
  - Exceeding the range of legal values results in OVERFLOW, a run-time error.

- The following operators are available for type `int` :
  `+` (addition)
  `-` (subtraction, negation)
  `*` (multiplication)
  `/` (integer division, where fraction is lost; result is an `int`)
  Example: `7 / 3 gives 2`
  `%` (remainder from division)
  Example: `7 % 3 gives 1`
  `== != > < >= <=` (comparisons: these take two values of type `int` and produces a `boolean` value)

An example: The Type `int`

Short-Hand Notation

- A very common expression is used to increment integer variables:

```
int i;
i = i+1;
```

- This is so common in computer programs that a short-hand notation is used:

```
i++;
```

- This will be very practical when defining loops.

Type `double`

- The following operators are available for type **`double`** :
  - **`+`** (addition)
  - **`-`** (subtraction, negation)
  - **`*`** (multiplication)
  - **`/`** (division in "real" numbers, result is a **`double`**)
  - **`> <`** (comparisons: these take two values of type **`double`** and return a **`boolean`** value)

- WARNING: Using **`==`** (or **`!= >= <=`**) to compare two values of type **`double`** is legal, but NOT recommended, because of the potential for round-off errors.
  - Instead of **`(a == b)`**, use
    `(Math.abs(a - b) < 0.001)`
    where **`Math.abs(x)`** returns |x|

Type `boolean`

- Only two literals: `true` and `false`
- The following operators are available for type `boolean`:

  `!` NOT (a unary operator, similar to a negative sign)

  `&&` AND

  `||` OR

  comparison : `==` `!=`

Type `char` (1)

- Characters are individual symbols, enclosed in single quotes
- Examples
  - letters of the alphabet (upper and lower case are distinct) `'A'`, `'a'`
  - punctuation symbol (e.g. comma, period, question mark) `','`, `'?'`
  - single blank space `' '`
  - parentheses `'('`,`')'`; brackets `'['`,`']'`; braces `'{'`,`'}'`
  - single digit (`'0'`, `'1'`, … `'9'`)
  - special characters such as `'@'`, `'$'`, `'*'`, and so on.

Type `char` (2)

- Each character is assigned its own numeric code:
  - ASCII character set (ASCII = American Standard Code for Information Interchange)
    - 128 characters
    - most common character set (if you speak American English ☺)
    - used in older languages and computers
  - UNICODE character set
    - over 64,000 characters (international)
    - includes ASCII as a subset
    - used in Java

Type conversion

- In general, one CANNOT convert a value from one type to another in Java, except in certain special cases.

- When a type conversion is allowed, you can ask for a type conversion as follows (this is called "casting"):

```
(double) 3 gives 3.0
(int) 3.5 gives 3          (note loss of precision!)
(int) 'A' gives 65         (this is the UNICODE value)
(char) 65 gives 'A'
```

- WARNING: Type conversions with unexpected results have resulted in serious software problems.
  - One such error caused the self-destruction of the Ariane 501 rocket in 1996.

- The best strategy: DON'T mix types or values, unless absolutely necessary!

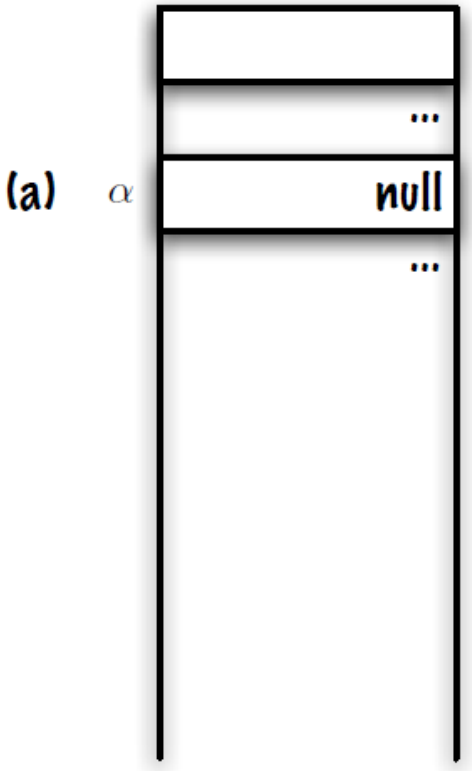
Agenda

1. Java programs:
2. What is a variable?
3. **What is a data types?**
    1. Primitive data types.
    2. **Reference data types**
        1. Example 1:Arrays
        2. Example 2: Strings
4. Methods in Java
5. Classes in Java

2nd type of data types: References Arrays

Example 1: Arrays

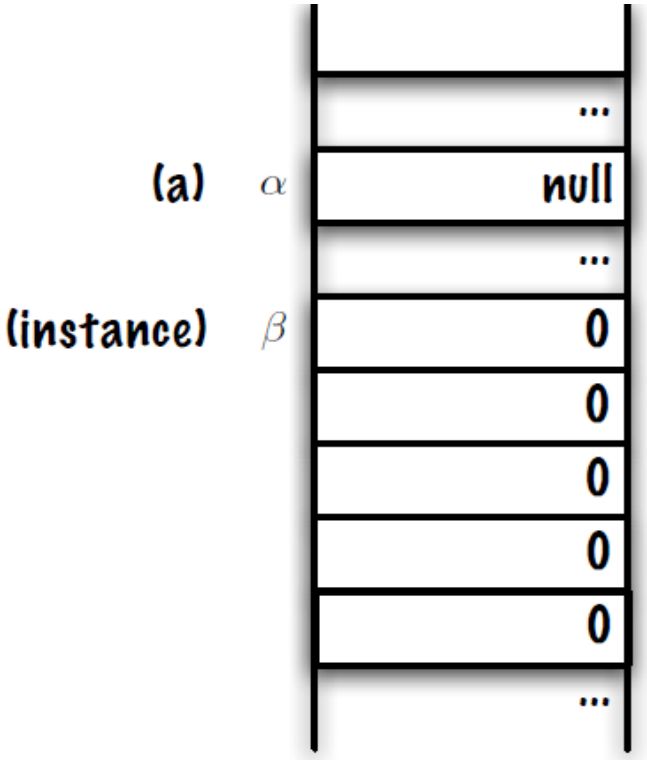
```
> int a [];
  a = new int[5];
```



⇒ The declaration of a reference variable only allocates memory to hold a reference (sometimes called pointer or address), *null* is a special value (literal), which does not reference an object

2nd type of data types: References Arrays

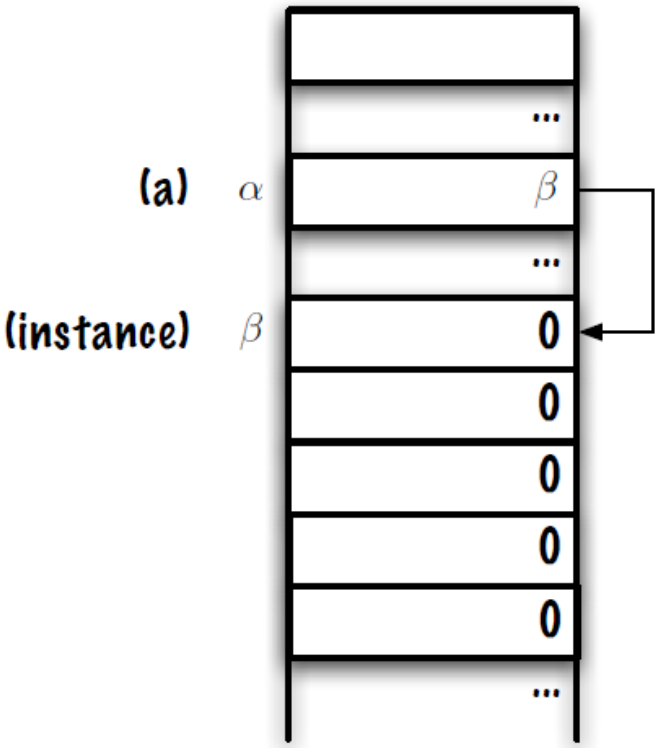
```
   int a[];
> a = new int[ 5 ];
```



⇒ The creation of a new instance, `new int[ 5 ]`, allocates memory to hold 5 integer values (and the housekeeping information). Each cell of the array is initialized with the default `int` value, which is 0.

2nd type of data types: References Arrays

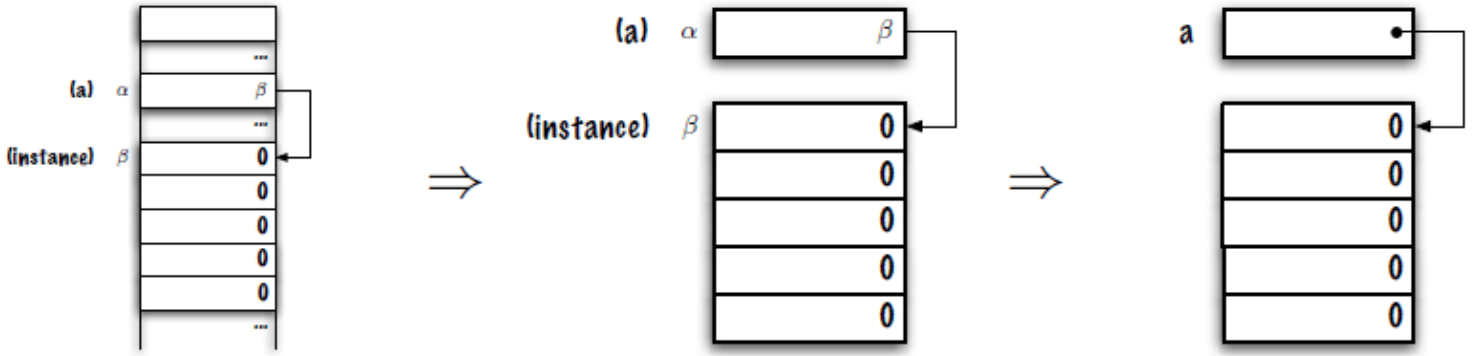
```
    int a[];
>   a = new int[ 5 ];
```



⇒ Finally, the reference of the newly created object is assigned to the location designed by the label **a**.

2nd type of data types: References Arrays

Because we don't know (and shouldn't care) about the actual memory layout, we often use memory diagrams such as the following,



2nd type of data types: References Arrays

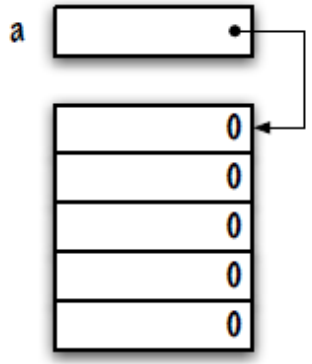
Memory diagrams:

In a memory diagram, I want to see:

- a box for every reference variable with an arrow pointing a the designated object
- a box for every primitive variable

```
int[] a;
a = new array[ 5 ];
```

⇒



2nd type of data types: References Arrays- summary

- An array reference variable is declared with the type of the members.
  - For instance, the following is a declaration of a variable of an array with members of the type double:

```
double[] anArray;
```

- When an array reference variable is declared, the array is NOT created.
- What we have is a **reference variable** that can point to an array.
  - `anArray` will contain the special value `null` until it is assigned a valid reference.

2nd type of data types: References Arrays- summary (2)

- To create the array in Java, operator **new** is used.
- We must provide the number of members in the array, and the type of the members: e.g. **new double[5]**
  - The number of members cannot be changed later.
  - The **new** operator returns a reference (address) that can be assigned to a reference variable, for example:

```
double[] anArray;
anArray = new double[5];
```

- Note: Creating an array initializes all elements to zeros (which translates to **0, null, '\0'** according to the type of the array)
- When an array is created, the number of positions available in the array can be accessed using a field called length with the dot operator.
- For instance, **anArray.length** has a value 5.

2nd type of data types: References Accessing array members

- Array members are accessed by indices using the subscript operator `[]`. The indices are integers starting from 0.
- For instance, if `anArray` is an array of three integers, then:
  - the first member is `anArray[0]`
  - the second member is `anArray[1]`,
  - and the third member is `anArray[2]`.
- The indices can be any expression that has an integer value.

  If an index is out of range, i.e., less than `0` or greater than `length-1`, a run-time error occurs.

Initializing array members

- Array members can be initialized individually using the indices and the subscript operator.

```
int [] intArray = new int[3];
intArray[0] = 3;
intArray[1] = 5;
intArray[2] = 4;
```

- Array members may also be initialized when the array is created:

```
int [] intArray;
intArray = new int [] { 3, 5, 4 };
```

2nd type of data types: References Accessing array members

- A STRING is a collection of characters.
  - There is NO primitive data type in Java for a string.
    - We will see later how to deal with strings in general.

- String literals (constants) can be used to help make your program output more readable.
  - String literals are enclosed in double quotes:

    `"This is a string"`

- Watch out for:
  - `"a"` (a string) versus `'a'` (a character)
  - `" "` (a string literal with a blank that has length 1) versus `""` (an empty string: a string literal of length 0)
  - `"257"` (a string) versus `257` (an integer)
  - “ ” « » are not valid quotes in Java!

2nd type of data types: References Strings

- Strings in Java are also accessed using reference variables.
  - They are similar to an array of characters.
  - EXCEPT:
    - You don't need to use `new` to create a string
    - You don't use `[]` to access the characters in the string.
- Example:

```
String message = "Hello World!";
System.out.println( message );
```

- There is a class (data type) `String` that provides many useful methods.
  - This means that Strings are objects (more on objects in the second half of the course).

Agenda

1. Java programs:
2. What is a variable?
3. What is a data types?
    1. Primitive data types.
    2. Reference data types
        1. Example 1:Arrays
        2. Example 2: Strings
4. **Methods in Java**
5. Classes in Java

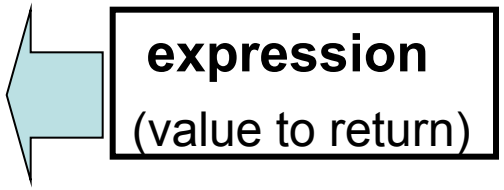
Java Methods

- the Java Method is a sub-program
- Every method in Java has
  - Return type: It specifies what is the type of the result of the method. If there is no result, this type is **`void`**.
    - For now always include keywords "public static" before the return type.
  - Name: To reference the method (same as the name of an algorithm).
  - Parameter list: It specifies the name and the type of the parameters, in order.
    - Note that the parameter list is a list of "variable declarations".
  - Body: An instruction bloc consisting of a sequence of Java instructions which is the translation of the algorithm body
    - Notice the form of the instruction bloc: the use of the braces { } and the indentation.

Method Template

```
// METHOD Name: Short description of what the
// method does, plus a description of
// variables in the parameter list
```


```
public static double avg3(int a, int b, int c)
{
   // DECLARE VARIABLES/DATA DICTIONARY
   // intermediates and
   // the result, if you give it a name

   // BODY OF ALGORITHM

   // RETURN RESULT
   return <returnedValue>;
}
```



A Java method "returns" a value

- A Java method may return no or one value.
  - It is not possible to return more than one value in Java.
  - This value may be a primitive type or a reference type.
  - For example, a method may return the reference to an array.

  - The Java method returns a "value", it does not assign a value to a variable
    - The call to a Java method can be interpreted as "reading the value of a variable" and as such, can be placed anywhere (i.e. expressions) a variable name is used.
    - In this course, to align with the algorithms, we shall assign the return value of a method call to a variable and in methods always declare a result variable
    - More details on method calls in Section 3

A method with return type **void**

- If a method does not return a value, i.e., return type **void**, then a call activates the method to do whatever it is supposed to do.

```
public static void print3(int x, int y, int z)
{
   System.out.println("This method does not return any value.");
   System.out.println( x );
   System.out.println( y );
   System.out.println( z );
}
```

- When the method is called by

  ```
  print3( 3, 5, 7 );
  ```

  it simply prints these arguments to the screen.

Arity of a A Java method

The **arity** of a method is simply the number of parameters; a method may have no parameter, one parameter or many.

A Java method

A **formal parameter** is a variable which is part of the definition of the method; it can be seen as a local variable of the body of the method.

```
int sum( int a, int b ) {
    return a + b;
}
```

⇒ **a** and **b** are formal parameters of **sum**.

A Java method

An **actual parameter** is the variable which is used when the method is called to supply the value for the formal parameter.

```
int sum( int a, int b ) {
    return a + b;
}
...
int midTerm, finalExam, total;
total = sum( midTerm, finalExam );
```

midTerm and **finalExam** are actual parameters of **sum**, when the method is called the value of the actual parameters is copied to the location of the formal parameters.

A Java method

Concept: call-by-value

In Java, when a method is called, the **values** of the actual parameters are **copied** to the location of the formal parameters.

When a method is called:

- the execution of the calling method is stopped
- an activation frame (activation block or record) is created (it contains the formal parameters as well as the local variables)
- the value of the actual parameters are copied to the location of the formal parameters
- the body of the method is executed
- a return value or (void) is saved
- (the activation frame is destroyed)
- the execution of the calling method restarts with the next instruction

A Java method

```
public class Test {

  public static void increment( int a ) {
    a = a + 1;
  }

  public static void main( String[] args ) {
    int a = 5;
    System.out.println( "before: " + a );
    increment( a );
    System.out.println( "after: " + a );
  }
}
```

What will printed?

```
before: 5
after: 5
```

"Standard" Java Methods

- Java offers "standard" classes and methods
  - For example methods for doing certain math functions
    - Used `Math.abs(a-b)` to implement |a-b|
    - To translate $\sqrt{x}$ use `Math.sqrt(x)`
  - Other methods exist to read from the keyboard and write to a terminal console
    - Shall explore such methods in just a moment.

Java Output

- To output a value to the console, we will use two Java methods from the class **`System.out`**:

  `System.out.println("aaa\nbbb")`

  `System.out.print(...)`

  - Method **`println()`** will append a new-line character to the output, while **`print()`** does not.

- Calls to these methods are unusual in that they will accept any type of argument or no argument.
  - Method call **`println()`** with no argument can be used to print a blank line.

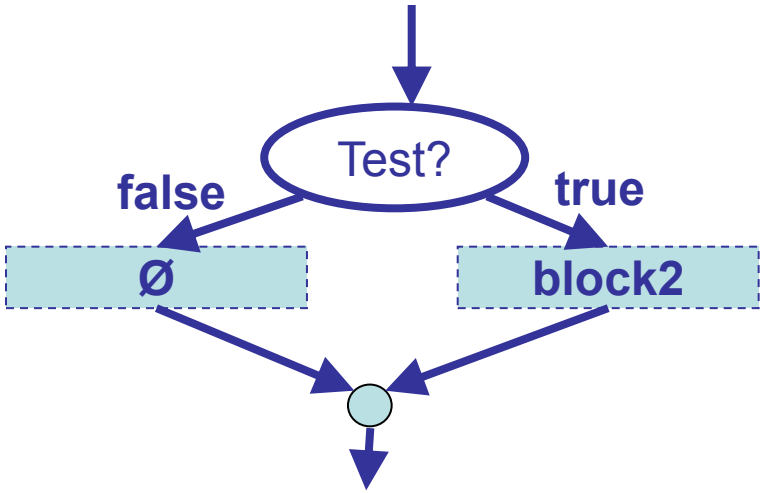
Program Template (To be Used for Assignments)

```
// Comments identifying you
// Comments describing what the program does
// and how it is used (e.g. what input is required)
import java.io.* ;

class PutYourClassNameHere
{
   public static void main (String args[ ])
   {
      // DECLARE VARIABLES/DATA DICTIONARY
      // PRINT OUT IDENTIFICATION INFORMATION
        System.out.println("ITI1121, Assignment x);
        System.out.println("Name: Grace Hoper, Student# ");
        System.out.println();

      // BODY OF ALGORITHM - Call to the method to solve problem
      // PRINT OUT RESULTS AND MODIFIEDS
   }
  // Put the method called by "main".
}
```

Java if Instruction - Option

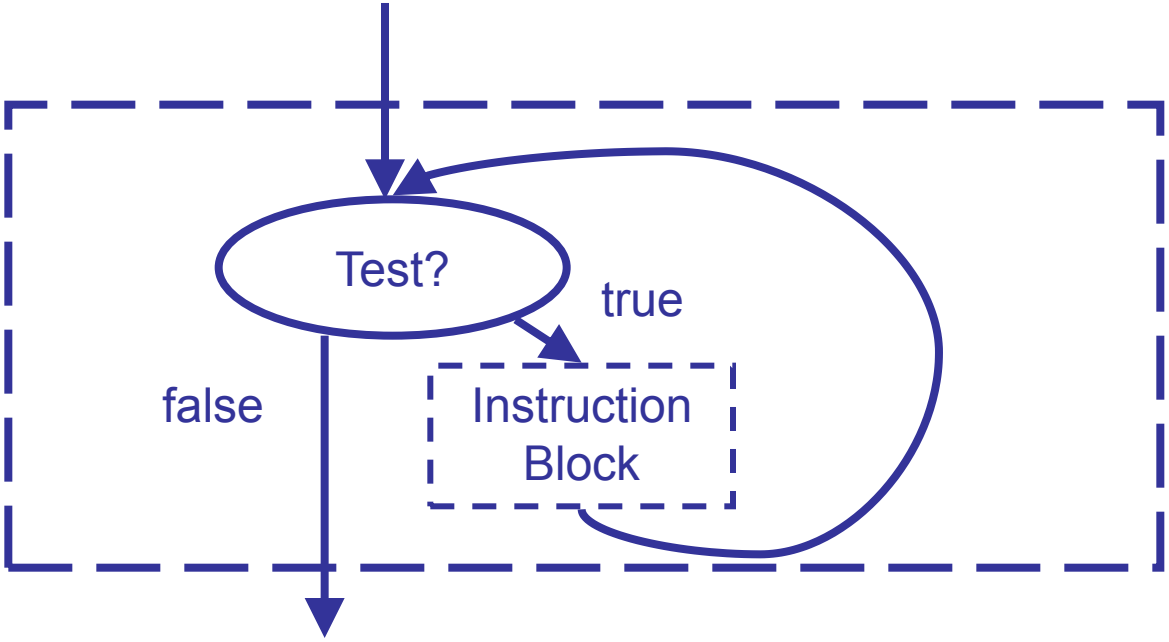
The else part of the instruction is optional. Thus in algorithms use only the empty statement block in the false part of the branch statement.

```
if (test)
{
    // Instructions
}
else
{
    // Instructions
}
```




```
if (test)
{
    // Instructions
}
```

Translating Loops to Java

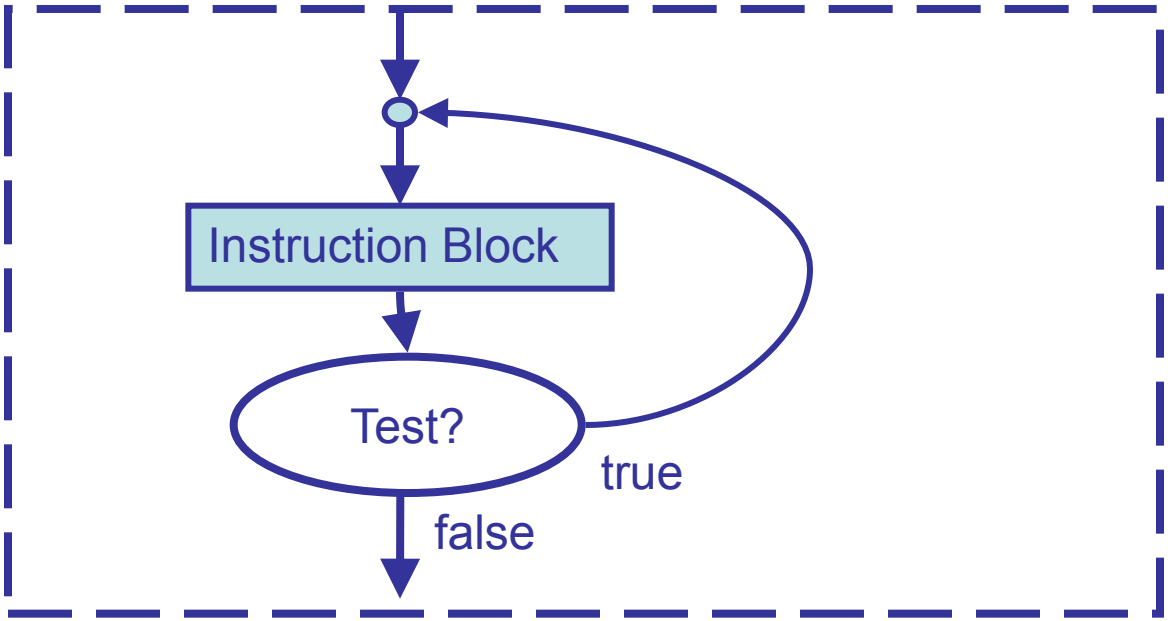


- Java

```
while (Test)
{
    // Instructions
}
```

Instruction block

The Post Test Loop Instruction - Translating to Java



- Java:

```
do
{
    // Instructions
}
while(test);
```

Instruction block

The FOR Loop

- Java provides another format of a loop, which is usually used when we know how many times the loop body is to be executed (definite loop).

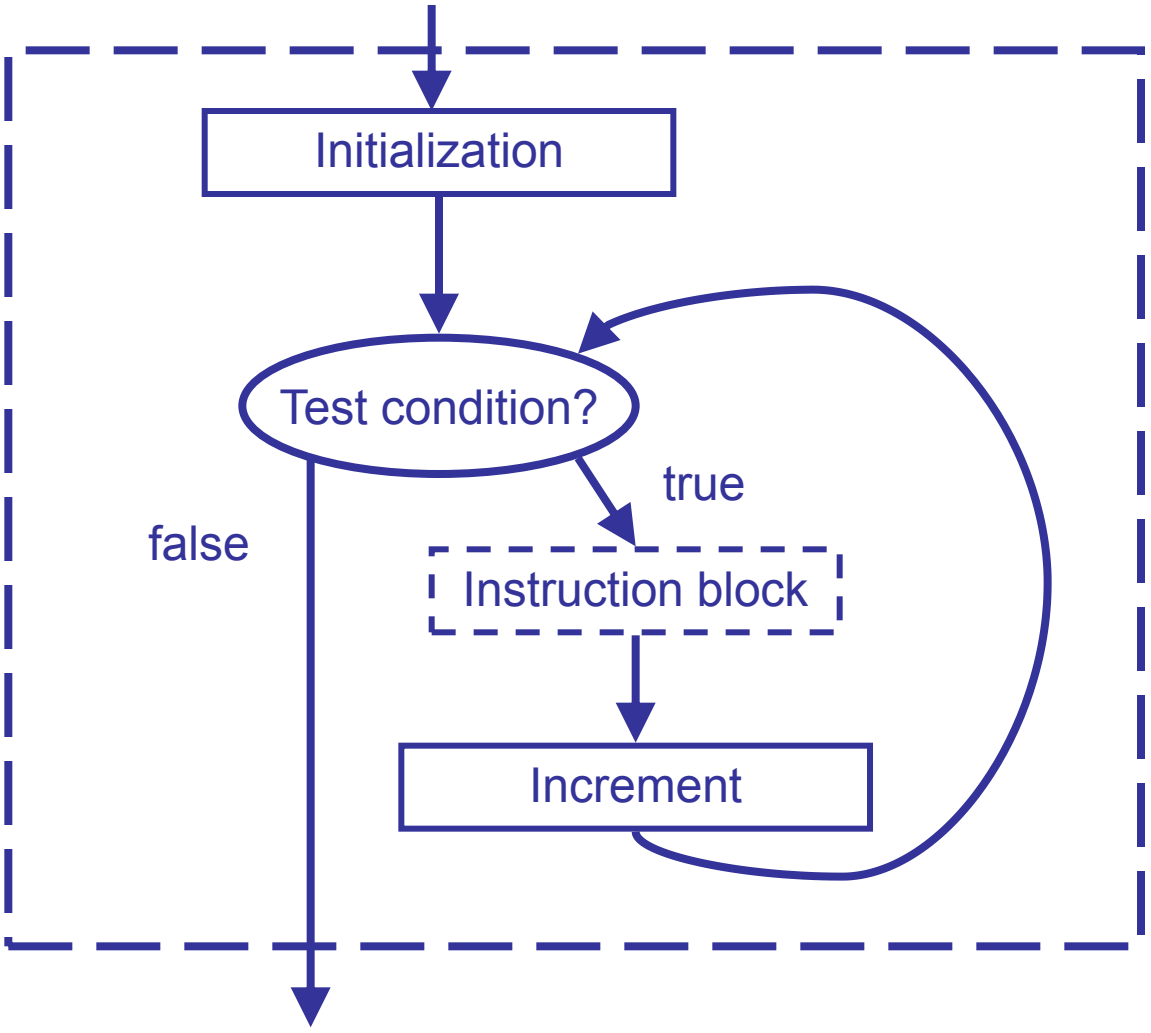
- The FOR loop has the following format:

```
for (<initialization>; <test_condition>; <increment>)
{
   // instructions
}
```

- In most cases, the initialization part initializes a counter, the test condition tests if the counter is within the limit, and the increment part modifies the counter.

- Any FOR loop can always be formed as a WHILE loop
  - It does not give us any extra capability.
  - However, the notation is often more convenient.

The FOR loop diagram



Using Multiple Methods

- The program starts with a **main** method (translated to a `main` method), which may read in some values and output the result. The **main** method calls one or more other methods.
  - In this way, the method `main` acts as a dispatcher.
  - Try to keep **main** as simple as possible – you should be able to tell what the overall program by examining **main**

Method Accessibility

- In Java,
  - Methods are collected inside a "class"
  - A Java program can be made up of many classes
- If a method is **`public`**, it can be called from anywhere in a program.
- If a method is **`private`**, it can only be called from inside the class where it is defined.

- There are two other levels of access: **`protected`** and "package", that are between public and private. (we will talk about them later.

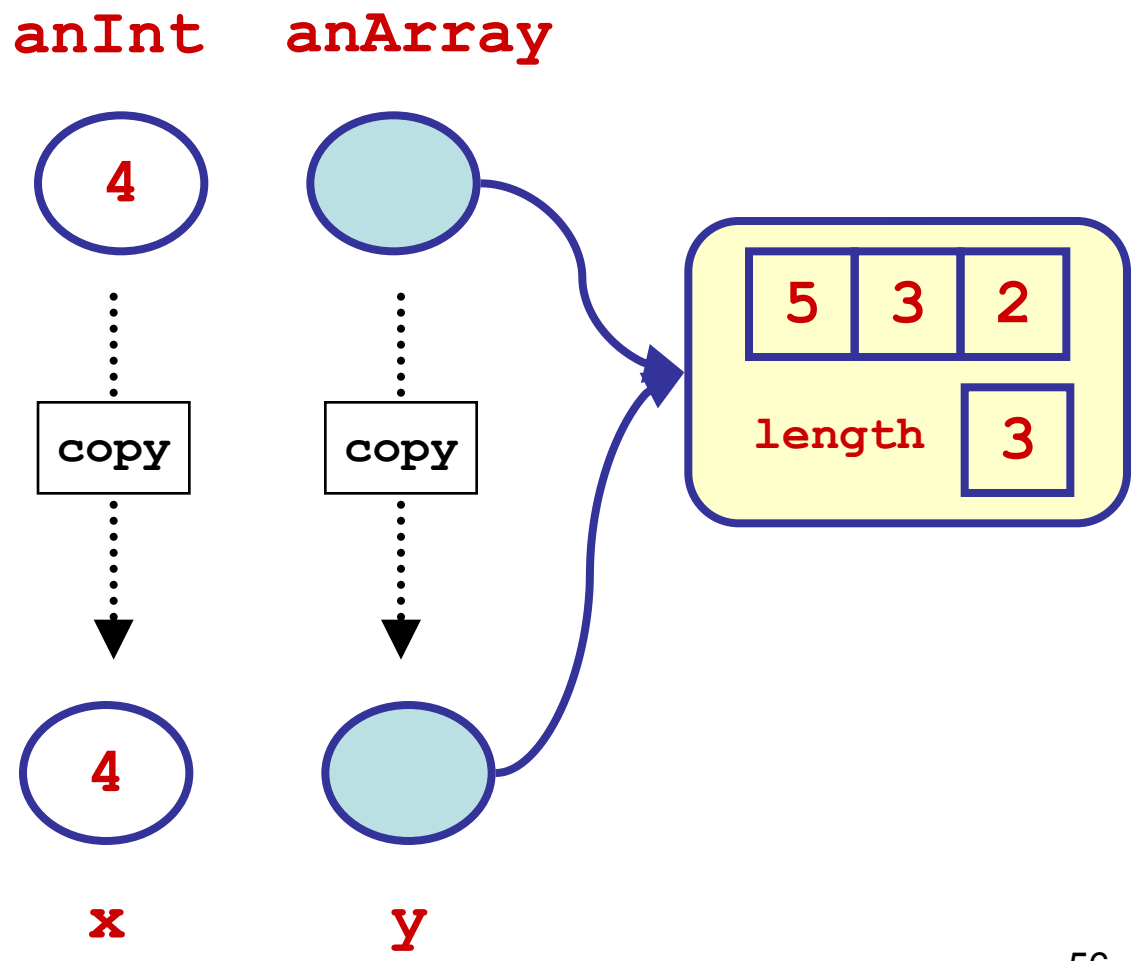
Arrays as Parameters

- An array is a reference type; i.e. is accessed using a reference variable.
- Arrays are not passed from one method to another method, it is the reference (i.e. the content of the reference variable) that is passed to a method (or can be returned by a method).
- The result is that there are (temporarily) two references to the same array.
- While we cannot modify the original reference variable, a called method can modify the contents of the array. These changes to the array contents will remain after the called method returns.
  - The copy of a variable of a primitive type is trashed when the method returns.
  - For an array, it is the copy of the reference variable that is trashed on return.

Passing primitive and reference types to a method

At caller:

`m(anInt,anArray);`

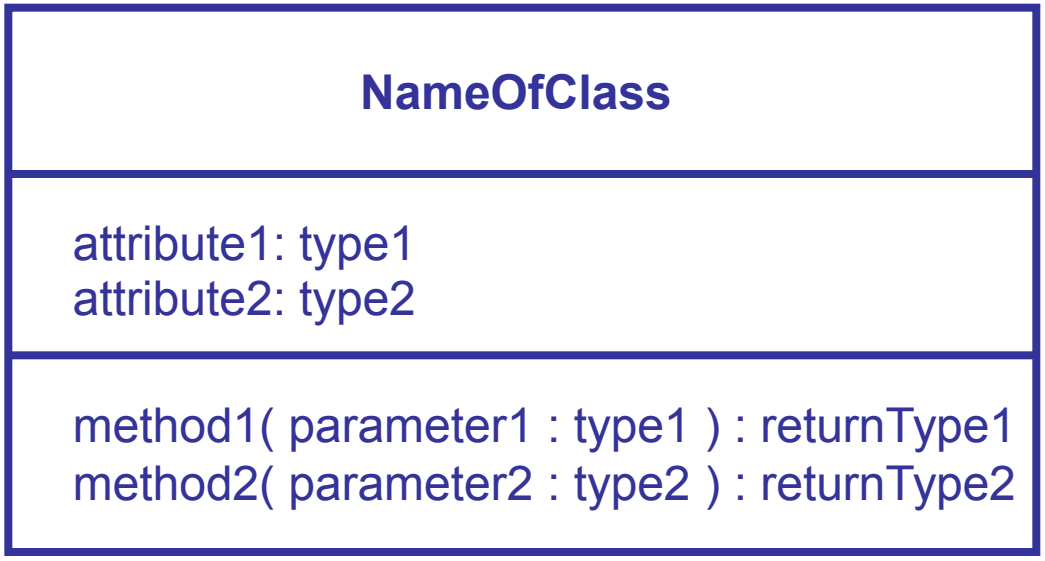


Agenda

1. Java programs
2. What is a variable?
3. What is a data types?
    1. Primitive data types.
    2. Reference data types
        1. Example 1:Arrays
        2. Example 2: Strings
4. Methods in Java
5. **Classes in Java**

3rd data type: Classes

Class diagram



"Attributes" are like the field variables in a record

- This form of diagram is from a notation called the "Unified Modelling Language" or UML

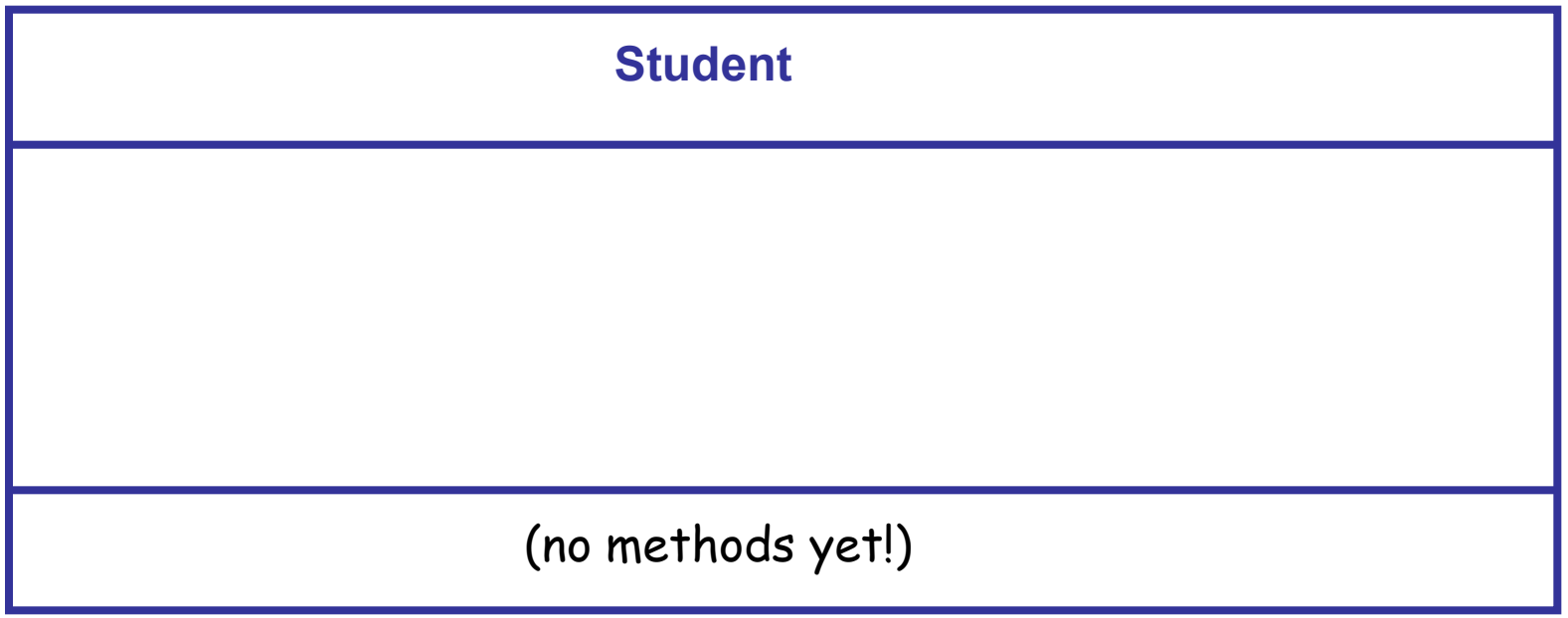
Classes and Objects

- A "class" can be used as a template to create objects with identical sets of attributes.

- The class can also contain methods (algorithm models) to perform calculations on the attributes of objects created from the class (and/or external data).

An Example

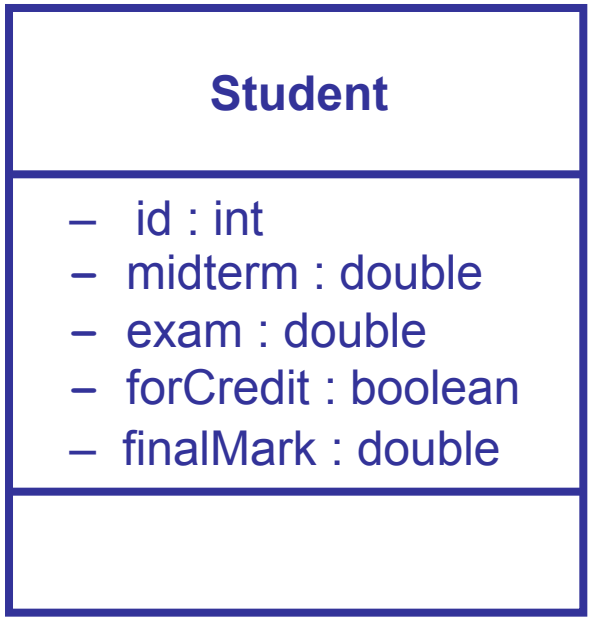
- For each student, we want to store their ID number, their midterm score, their exam score, and whether or not the student is taking the course for credit. *We shall deal with the final mark later.*



Translation to Java

```
public class Student
{




   // methods
}
// Declare aStudent reference Variable

// Create a Student object referenced by aStudent
```

Example: Student class- adding variables



- The - in front of the variable indicates that the attribute is private.
- By declaring a field to be private, only methods declared inside the class are allowed access to the field value (either for viewing the value, or changing the value)

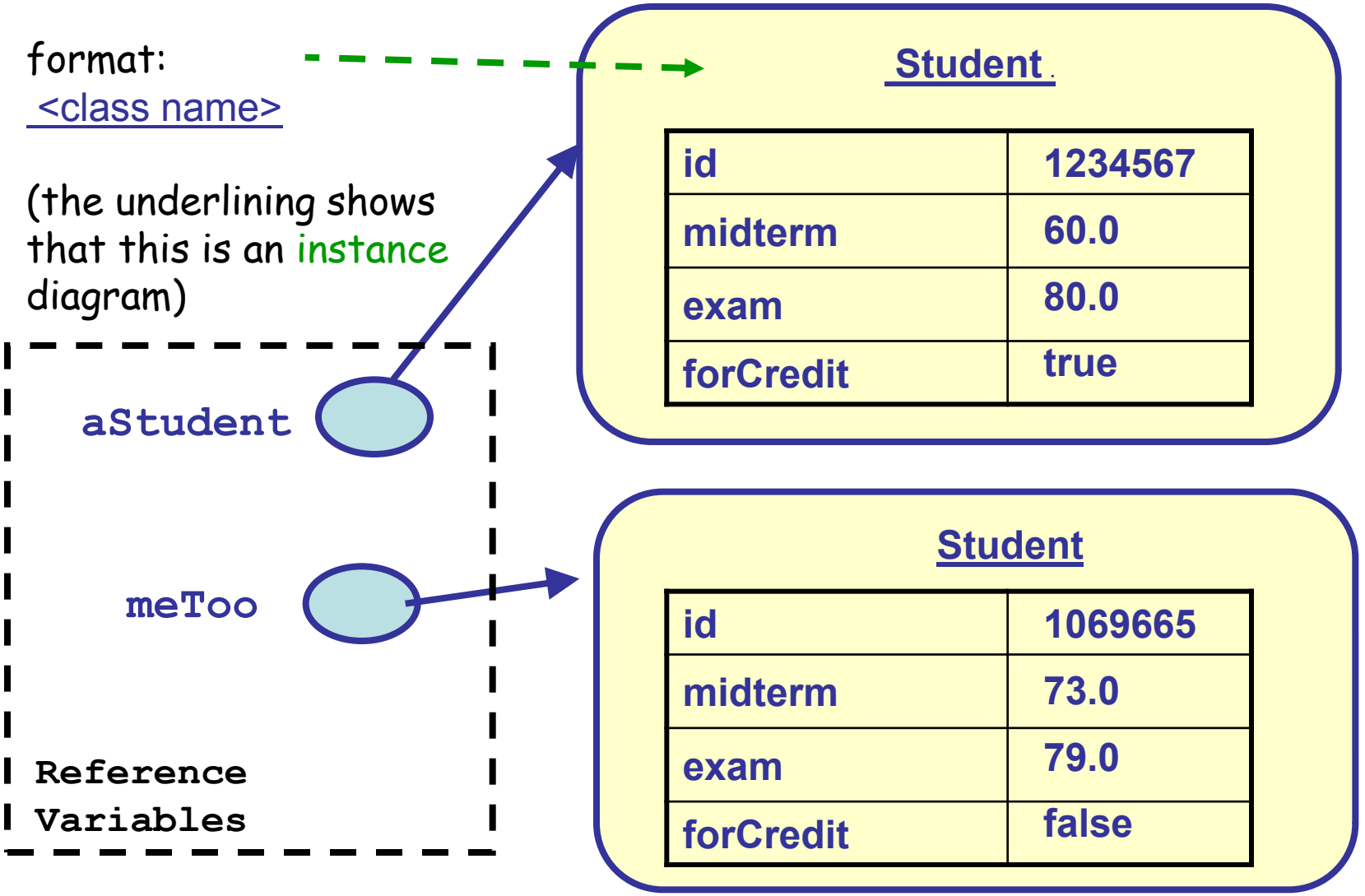
How to use the class

```
Student aStudent;           // declare reference variable
aStudent = new Student();   // create new object
aStudent.id = 1234567;
aStudent.midterm = 60.0;
aStudent.exam = 80.0;
aStudent.forCredit = true;

Student meToo;
meToo = new Student();
meToo.id = 81069665;
meToo.midterm = 73.0;
meToo.exam = 77.0;
meToo.forCredit = false;
```

How to use the class

format:
<class name>

(the underlining shows that this is an instance diagram)



Creating the Student Class

```
public class Student
{
    // were previously public
    private int id;
    private double midterm;
    private double exam;
    private boolean forCredit;
    private double finalMark;
   // methods
}

Somewhere in your main program ....
Student aStudent = new Student();
```

Information Hiding

- To ensure relative independence relative to other parts of your program (which helps reduce the effort of maintenance), fields are (almost) always **`private.`**
- This information hiding is also called data abstraction and also encapsulation).

- The private fields and methods cannot be accessed directly but only from methods in the class.

- If (and only if) necessary, can define a few public methods to allow other parts of the program to access fields.

- The public methods represent the interface of the class relative to other parts of the program.

How do we use the class

- If we try the following:

```
Student aStudent = new Student();
aStudent.id = 1234567 ; // error!
```

  the compiler returns an error since access to id is no longer allowed from outside the class.

- However, we can create additional access methods in the class `Student`:
  - "accessor": requests to see the value of a private field.
  - "modifier": requests to modify the value of a private field.

Accessors and Modifiers

- Accessor
  - A public instance method (called using a reference to an object);
  - Returns the value of the field of the object;
  - Has no parameters;
  - Often called `getFieldName` (also called a getter method).

- Modifier
  - A public instance method (called using a reference to an object);
  - Assigns a value to a field;
  - Accepts values in a parameter of the same type as the field;
  - Often called `setFieldName` (also called setter method).

Accessors and Modifiers

- Examples for the forCredit field in the class:

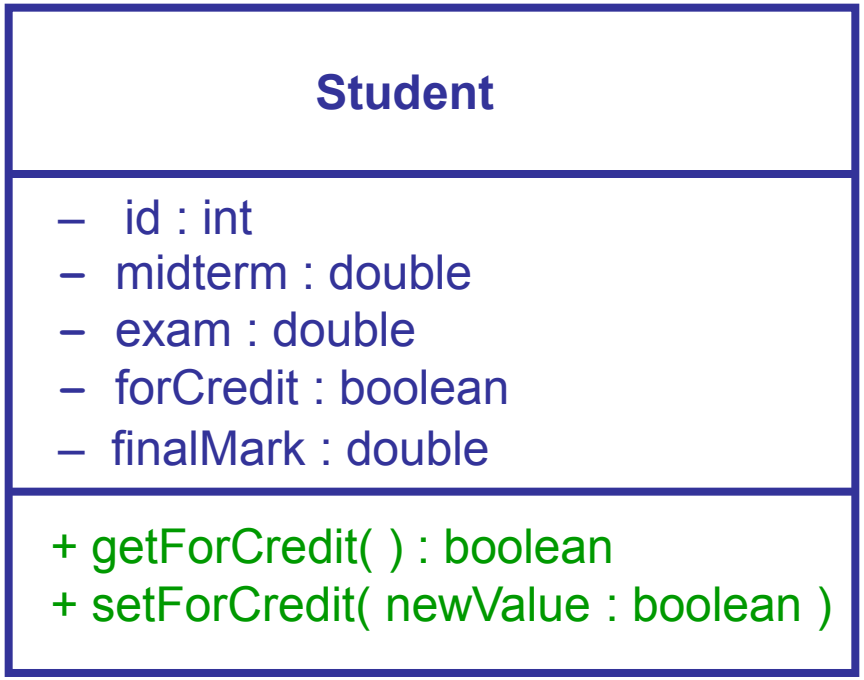
boolean getForCredit( )

  - method to return the value of forCredit
  - the + indicates that the method has public visibility
  - the return type is boolean, and in UML notation, appears at the end of the method.

void setForCredit(newValue : boolean)

  - method to change the value of forCredit
  - one parameter newValue, of type boolean
  - no return value

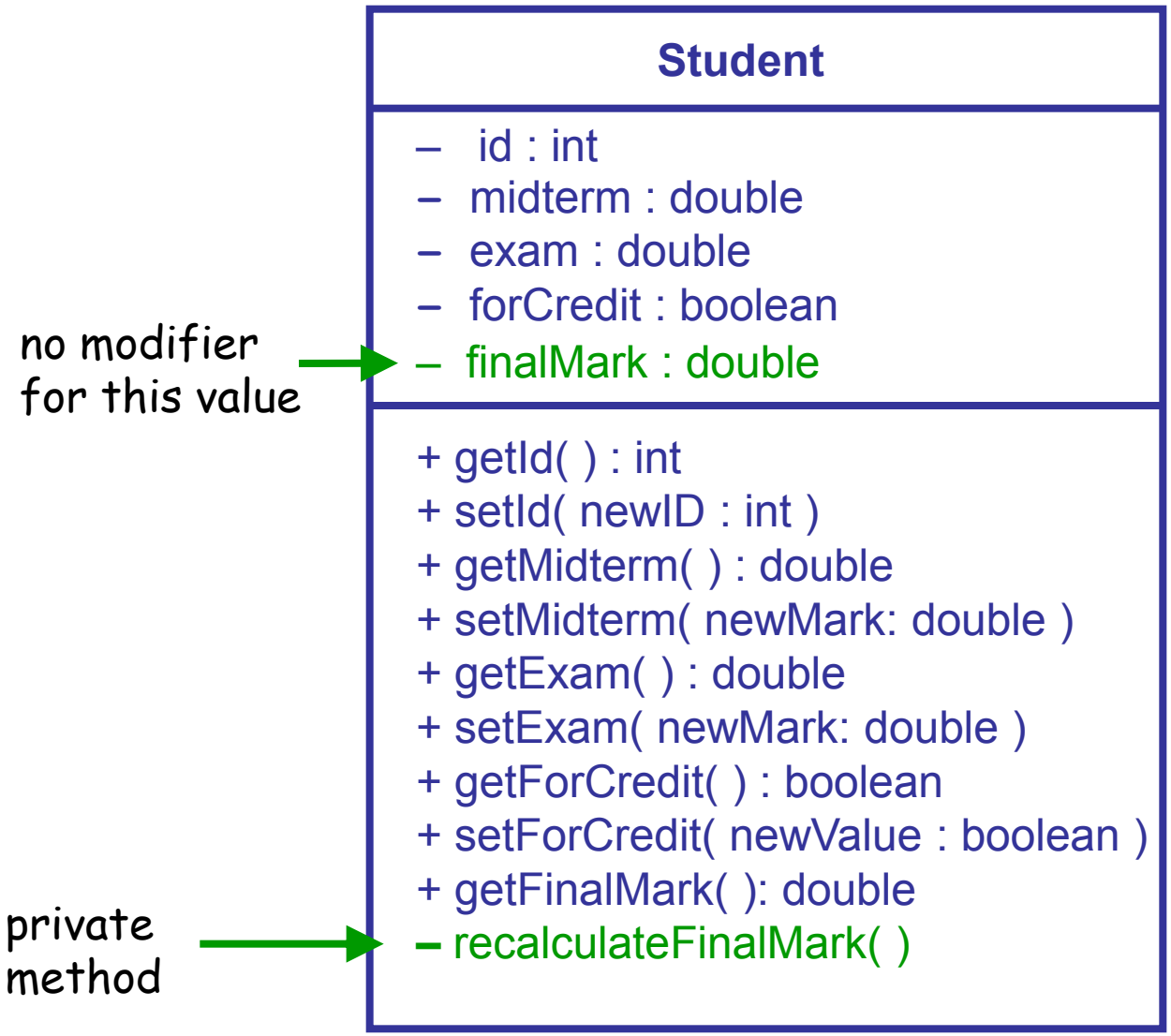
Class diagram with accessors and modifiers



Back to Information Hiding

- To implement our strategy of hiding the finalMark field, we can do the following:
  - We will provide an accessor method for finalMark, but NOT a modifier method.
  - We can provide a method recalculateFinalMark() to recalculate the final mark if the midterm or exam marks are changed.
  - The modifier methods setMidterm() and setExam() will call recalculateFinalMark() so that they automatically update the final mark.
- We should also restrict access to recalculateFinalMark() because it isn't meant for use outside the class.

Student class with Information Hiding



Translation to Java

```
public class Student
{
   // Attributes

   private int id;
   private double midterm;
   private double exam;
   private boolean forCredit;
   private double finalMark;

   // Methods

   public int getId()
   {
      // insert code here
   }
   public void setId( int newId )
   {
      // insert code here
   }
   public double getMidterm()
   {
      // insert code here
   }
   public void setMidterm( double newMark )
   {
      // insert code here
   }
   // continued at right
```

```
   // continued from left side

   public double getExam()
   {
      // insert code here
   }
   public void setExam( double newMark )
   {
      // insert code here
   }
   public boolean getForCredit()
   {
      // insert code here
   }
   public void setForCredit( boolean newValue )
   {
      // insert code here
   }
   public double getFinalMark()
   {
      // insert code here
   }

   private void recalculateFinalMark()
   {
      // insert code here
   }
} // end of class Student
```

Calling Java Accessor and Modifier Methods

- Again, use the dot operator (.)
- The following code causes errors, why?

```
Student aStudent = new Student();
aStudent.id = 1234567;                  // error here!
int myId = aStudent.id;                 // error here!
System.out.println( myId );
```

- The compiler enforces the private access to `id` .

- Solution: Instead, use the modifier and accessor methods.

```
Student aStudent = new Student();
aStudent.setId( 1234567 );          //ok!
int myId = aStudent.getId( );       //ok!
System.out.println( myId );
```

Implementing Java accessors and modifiers

```
public class Student   // not all attributes/methods shown!
{
    // attribute
    boolean forCredit;
    // … other attributes declared

    // accessor:  return the requested value
    public boolean getForCredit()
    {
        return this.forCredit ;
    }

    // modifier: save the requested value in object's
     // attribute

    public void setForCredit( boolean newValue )
    {
        this.forCredit = newValue;
    }
    // …other methods are similar
}
```

Where did `this` come from?

- When the fields of our Student class were public, we distinguished between the same field in two record objects with the variable name and the dot operator:
  - `aStudent.forCredit` versus `meToo.forCredit`
- Likewise, when a method inside the class wants to work with "the value of the field for the object on which I was called", `this` refers to the called object.
- During the call `aStudent.getForCredit()`, `this` is a reference to `aStudent`
  - … and so `this.forCredit` is `aStudent.forCredit`, which is true.
- During the call `meToo.getForCredit()`, `this` is a reference to `meToo`
  - … and so `this.forCredit` is `meToo.forCredit`, which is false.

Methods Operate on Object Data

- Adding a method to a class such as Student provides operations on the data in the objects created with the class (user defined types).

- Note that we have called the method getForCredit without parameters in two different cases
  - Each different case calls to the methods are associated to different objects and thus we get different results.

- An analogie:
  - With integers: 3 + 5 and 4 + 3; the same operation (+) is invoked, but with different values which gives different results
  - With Students: the same operation (getForCredit), but on different objects $\Rightarrow$ different results

Implementing Information Hiding

- The following implements our strategy where the final mark can only be changed by modifying the midterm or exam values.

```
public class Student
{
    // attributes and other methods would go here
    public void setMidterm( double newValue )
    {
        this.midterm = newValue;
        this.recalculateFinalMark( );
    }

    public void setExam( double newValue )
    {
        this.exam = newValue;
        this.recalculateFinalMark();
    }

    private void recalculateFinalMark()
    {
        this.finalMark = 0.2 * this.midterm + 0.8 * this.exam;
    }
}
```

Benefits of Information Hiding (1)

- One of the most common causes of problems historically has been when all parts of a program have access to all program variables.
  - For example, when someone makes a change to a large program, the new code may make changes to data that some other part of the program assumed would not be modified.

> "Successful software always gets changed." - *F. Brooks*

- With information hiding, we can keep the code better partitioned so that changes will be less likely to cause unwanted side effects.

Benefits of Information Hiding (2)

- We can also make changes inside a class that will not affect users of the class.

- Example: Suppose we decide that the finalMark field really doesn't need to be stored in the Student class.
  - Instead, we can calculate the final mark when anyone asks for it:

```
public double getFinalMark()
{
    return 0.2 * this.midterm + 0.8 * this.exam;
}
```

  - This means we can remove the method recalculateFinalMark( ), and the calls to it in setMidterm( ) and setFinal( ).

- Making these changes will not affect any user of the class:
  - For example, meToo.getFinalMark( ) still behaves as it did before.
  - Since recalculateFinalMark( ) was private, code outside the class was not able to call this method, and therefore it can be safely removed.
- So we don't have to change any code outside the class!

`this`, again

- In most cases, we don't actually have to use `this` to refer to the object on which a method is called.
  - Inside the Student class:
    - `exam` can be used instead of `this.exam`.
    - `recalculateFinalMark()` can be used instead of `this.recalculateFinalMark()`.

- There are 2 occasions when we really do need `this`:
  1. An object wants to pass itself as a parameter to a method of another class.
  2. An object wants to return a reference to itself as the result of a method.

Using Constructors

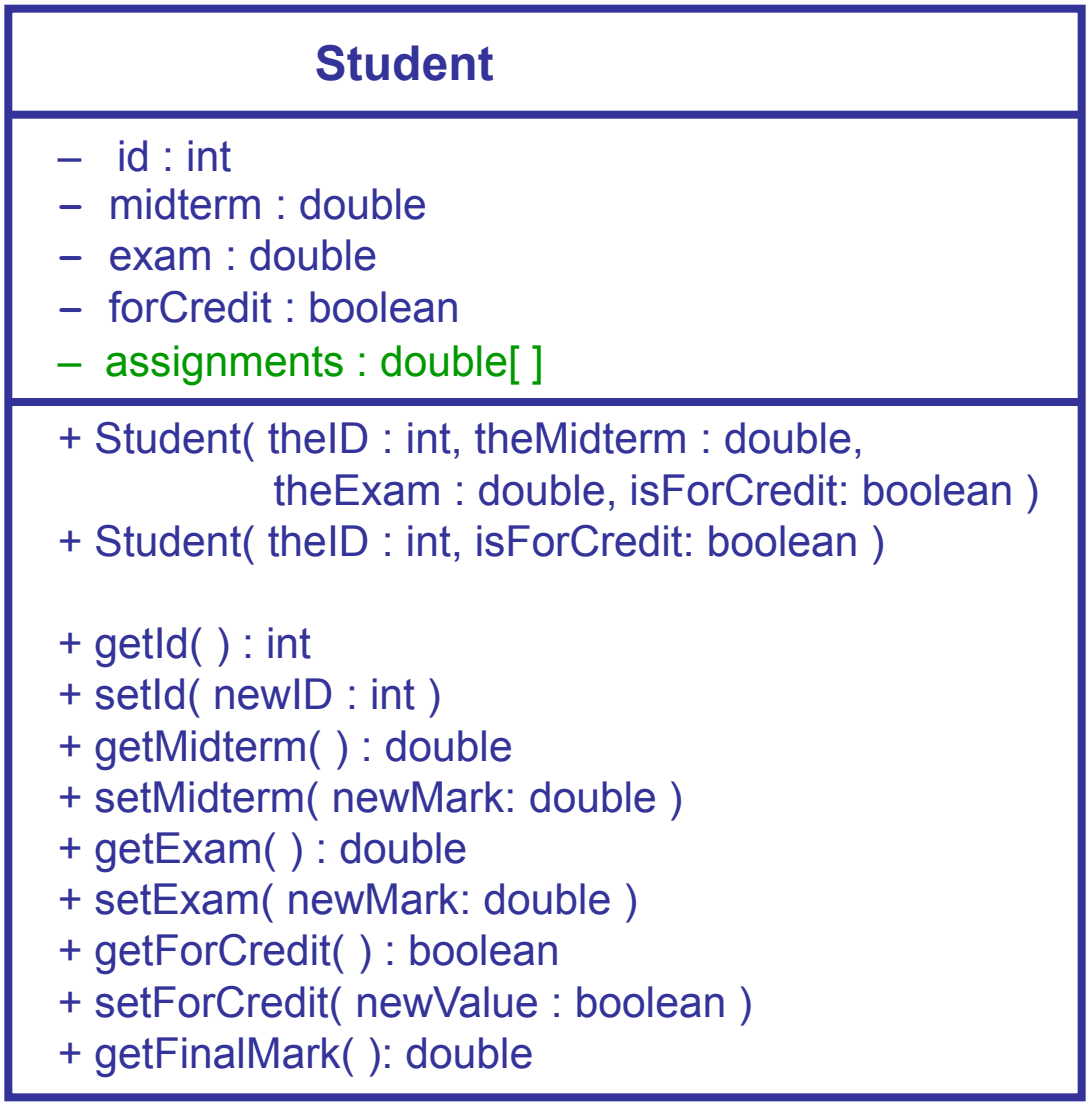
- **Show how objects are created and referenced by the following main method.**

```
public class Section11
{
   public static void main(String [] args)
   {
        Student aStudent; // reference variable
        Student meToo;    // another reference variable
        Student bStudent; // a third reference variable
                  •
                  •
        aStudent = new Student(1234567,60.0,80.0,true);
        meToo = new Student(7654321,true);
        bStudent = aStudent;
          •
          •
          •
   }
}
```

Using Constructors

Reference Variables

Array Fields in Classes



- A field of a class may have any type. In particular, a class may have a field of an array type (reference variable).
- Add an array of double to class Student representing assignment marks:
- Remember, arrays are not created automatically! The array assignments will have to be created after a Student object is created.

Array Fields in Classes

```
public class Student
{
    private int id ;
    private double midterm ;
    private double exam ;
    private boolean forCredit;
    private double[] assignments;

    // methods
}
```

- The array reference variable **assignments** contains the value **null**.

Array field initialization

- Here is a constructor that creates and initializes an array in an object. The constructor has a parameter that is the number of assignments.

```
public Student( int numberOfAssignments )
{
   this.id = 0;
   this.midterm = 0.0;
   this.exam = 0.0 ;
   this.forCredit = false;
   this.assignments = new double[numberOfAssignments];

   // loop to initialize each item in array
   int index;
   for ( index=0; index < numberOfAssignments; index = index+1 )
   {
      this.assignments[index] = 0.0;
   }
}
```

Translation to Java

```
public class <Class Name>
{
    // Declaration of Variables
    // public <type> <name>;

    // Methods
}
```

- The class is a "template" for how to construct objects.
  - Objects must be created using the **new** statement
  - Objects are referenced with a reference variable

```
// Declaring the reference variable
<Class Name> refVar;
// creating the object
aStudent = new <Class Name>( );
```

Student Class

- For each student, we want to store their ID number, their midterm score, their exam score, and whether or not the student is taking the course for credit. *We shall deal with the final mark later.*

Student
(no methods yet!)

Good to know....

Comments

- A comment is an explanation of the meaning of your variable or code.
- Java comments have (mainly) two forms:
  - A comment starting from `//` to the end of a line, such as

```
int size = 30; // number of students in a class
```

  - A comment between `/*` and `*/`, such as

```
/* this program calculates the sum of N
 * numbers and displays the result.
 */
```

- Comments are used to help people to read the program, and are ignored by the computer.

"Being forced to write comments actually improves code, because it is easier to fix a crock than to explain it."
-- G. Steele

Useful `String` methods

- Suppose we have

```
String message = "Hello World!";
```

  Then:
  - To find the length of a string:

```
int theStringLength = message.length();
```

  - To find the character at position i (numbered from 0):

```
int i = 4;
char theChar = message.charAt( i );
```

- To change any primitive data type to a `String` :

```
int anInteger = 17;
String aString = String.valueOf( anInteger );
// works for int, double, boolean, char
```

- To append one string after another (concatenation):

```
String joinedString = string1 + string2;
```

Comparing Strings

- A `String` is a reference type and so they are NOT compared with `==`.
- The `String` class has a method `compareTo()` to compare 2 strings.
  - The characters in each string are compared one at a time from left to right, using the collating sequence.
    - The comparison stops after a character comparison results in a mismatch, or one string ends before the other.
      - If str1 < str2, then `compareTo()` returns an `int` < 0
      - If str1 > str2, then `compareTo()` returns an `int` > 0
    - If the character at every index matches, and the strings are the same length, the method returns `0`

String Concatenation

- Strings can be CONCATENATED (joined) using the + operator:
  - `"My name is " + "Diana"` gives `"My name is Diana"`

- String values can also be concatenated to values of other types with the + operator.
  - `"The speed is " + 15.5` gives `"The speed is 15.5"`
  - Because one of the values for the + operator is a `String`, the double is converted to a `String` value `"15.5"` before doing the concatenation.