Accelerating Information Technology Innovation

Lecture 1: Java Basics

http://aiti.mit.edu

Recap - Teaching Style

- Emphasis on self-learning:
  - We will encourage you to discover your own answers
  - The most important skill you will ever learn
- Emphasis on participation:
  - Ask questions during lecture
  - Provide constructive criticism
  - Suggest course topics
  - Interrupt if we use jargon or idioms

Recap - Self-Learning

- Use MIT's OpenCourseWare website to teach yourself Java
- Website: http://ocw.mit.edu
- ebooks
- Why self-teach?
  - Move beyond the course curriculum
  - Develop a more advanced final project
  - We are here to help!

Recap - Student Evaluation

- There are no tests!
- Students will be evaluated on labs and projects:
- Labs:
  - Design/Code
  - Output
  - Post-lab interview
- Projects:
  - Idea
  - Milestone Presentations
  - Demo

Recap - Collaboration

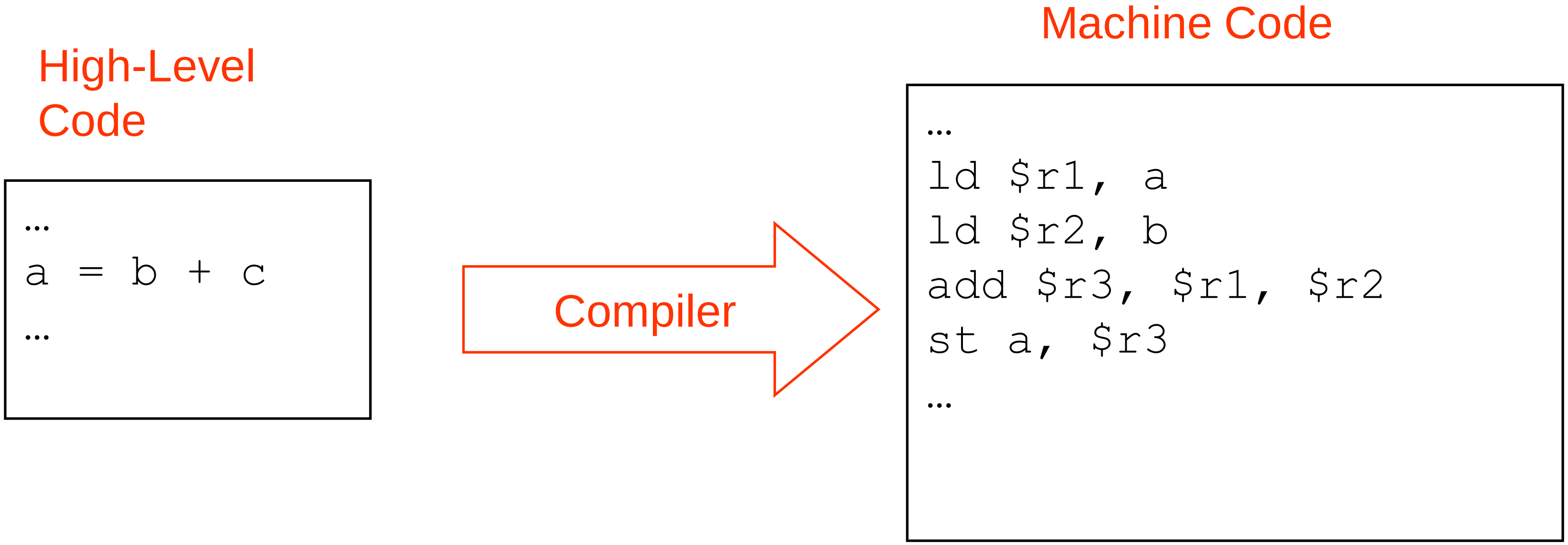
- Students are encouraged to collaborate on labs and projects.

- However, copying code without understanding is not allowed.

- Zero tolerance
  - If found copying, .. Well, we are not sure if you belong in the class. Its always better to ask for clarification than to copy!!

Starting Point - Compiler

- A program that translates a programming language into machine code is called a *compiler*



High-Level Code

```
…
a = b + c
…
```

Compiler

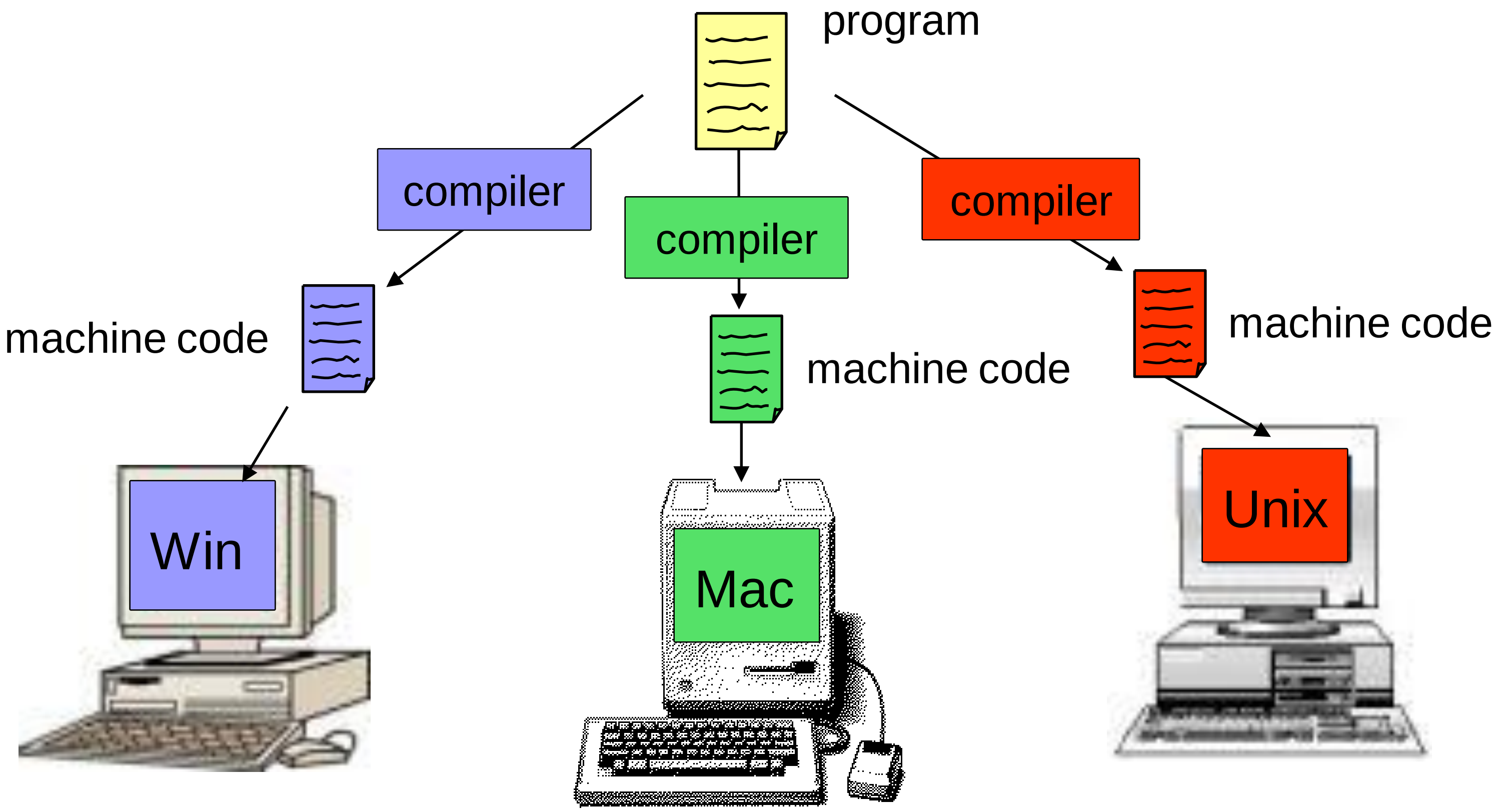
Machine Code

```
…
ld $r1, a
ld $r2, b
add $r3, $r1, $r2
st a, $r3
…
```

- Typically, we must have a compiler for each operating system/machine combination (*platform*)

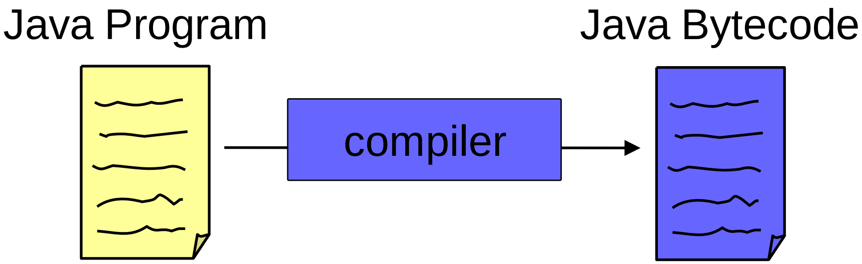
Compiling Computer Programs

- Because different platforms require different machine code, you must compile programs separately for each platform, *then* execute the machine code.



The Java Compiler is Different!

- The Java compiler produces an intermediate format called *bytecode.*

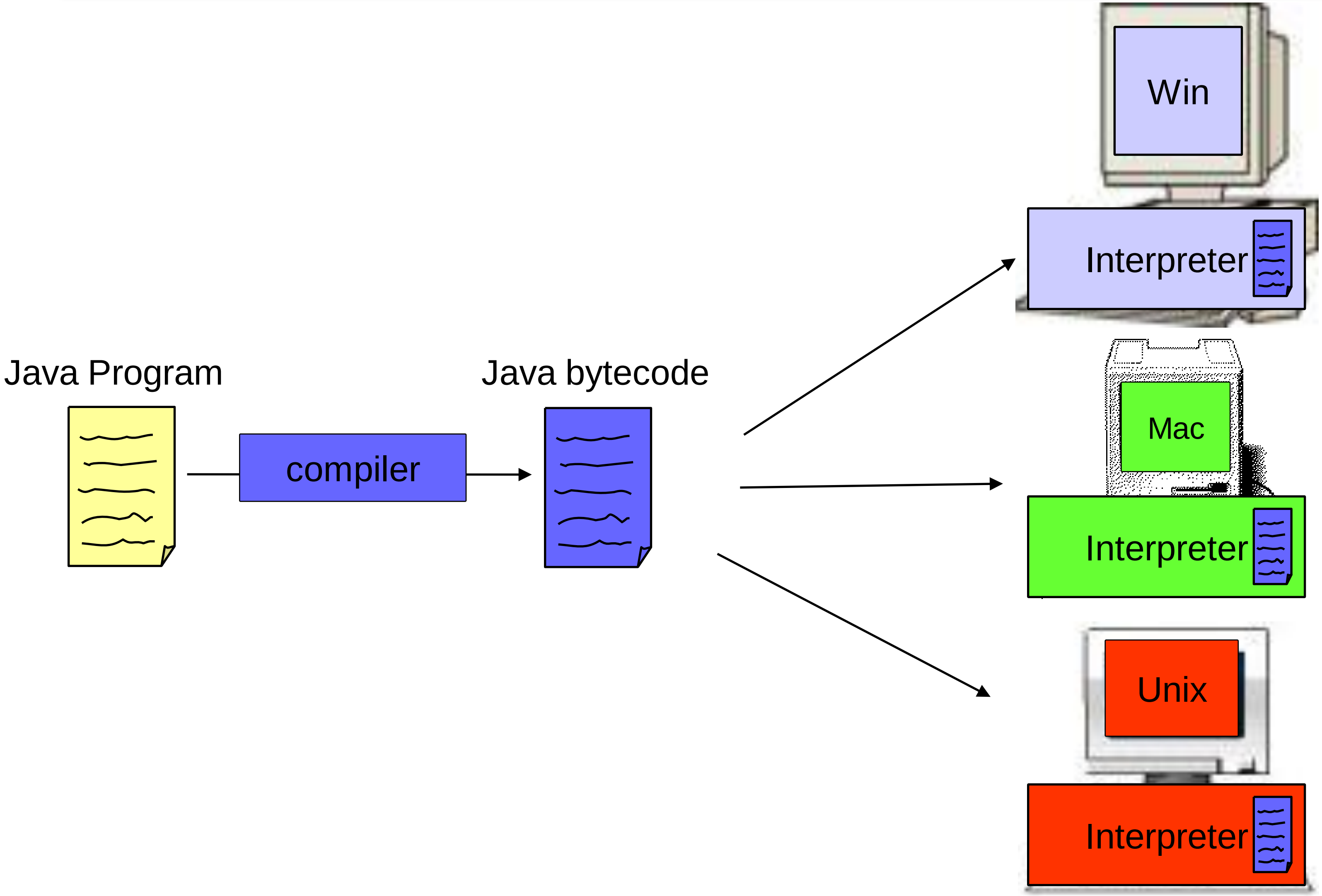


- Bytecode is not machine code for any real computer.
- Bytecode is machine code for a model computer.
  - This model computer is called the *Java Virtual Machine.*

Java Interpreter

- A Java *Interpreter* is required to execute the bytecode on a real computer.
- A Java Interpreter converts the bytecode into machine code.
  - As the program executes
  - *Simulate* the execution of the Java Virtual Machine on the real computer
- You can run bytecode on any computer that has a Java Interpreter (JRE) installed!
  - Only have to compile once
  - Can distribute the same bytecode to everyone

The Java Approach



Advantages of Using Java

- Once a Java program is compiled you can run the bytecode on any device with a Java Interpreter.
  - Because you do not have to recompile the program for each machine, Java is *device independent.*

- Java is safe. The Java language and compiler restrict certain operations to prevent errors.
  - Would you want an application to have total control of your phone?
    - Make calls, send SMS messages?

- Java standardizes many useful structures and operations such as lists, managing network connections, and providing graphical user interfaces

Disadvantages of Using Java

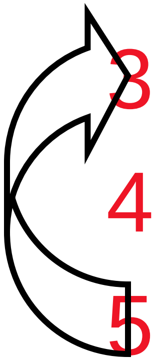
- Running bytecode through an interpreter is not as fast as running machine code
  - But this disadvantage is slowly disappearing

- Using device specific features (e.g., bluetooth) is difficult sometimes because Java is device-independent.

- In order to run a Java program on multiple devices, each must have a Java Interpreter
  - Ex: most Nokia phones come with Java Interpreter

Programming Methodology

1. Specify and analyze the problem
   - Remove ambiguity
   - Decide on inputs/outputs and algorithms
2. Design the program solution
   - Organize problem into smaller pieces
   - Identify existing code to reuse!



3. Implementation (programming)
4. Test and verify implementation
5. Maintain and update program

Writing Good Code

- A program that meets specification is not necessarily good.
- Will you be able to make changes to it?
  - Will *you* understand it after some time?
- Others might need to look at your code
  - Can they understand it?
- Write your program so that is easy to understand and extend!
  - Spend extra time thinking about these issues.

Example Code: Comments

```
/* The HelloWorld class prints “Hello,
World!” to the screen */
public class HelloWorld {
  public static void main(String[] args) {
    // Prints “Hello, World!”
    System.out.println("Hello, World!");
   // Exit the program
   System.exit(0);
  }
}
```

Comments

- *Comments* are used to describe what your code does as an aid for you or others reading your code. The Java compiler ignores them.
- Comments are made using `//`, which comments to the end of the line, or `/*  */`, which comments everything inside of it (including multiple lines)
- Two example comments:
  - `/* The HelloWorld class prints “Hello, World!” to the screen */`
  - `// Prints “Hello, World!”`

Comments on Commenting

- You may collaborate on software projects with people around the world who you’ll never meet
- Should be able to figure out how code works by reading comments alone
- Anything that is not self-evident needs a comment
- 50% of your code might be comments
- Coding is easy, commenting is not

Less Talk, more play!

Lab Section 1

Accelerating Information Technology Innovation

Variables and Operators

http://aiti.mit.edu

Declaring Variables in Java

```
type name;
```

- Variables are created by declaring their type and their name as follows:

- Declaring an integer named “x” :
  - `int x;`
- Declaring a string named “greeting”:
  - `String greeting;`
- Note that we have not assigned values to these variables

Java Types: Integer Types

- Integer Types:
  - int: Most numbers you will deal with.
  - long: Big integers; science, finance, computing.
  - short: Smaller integers. Not as useful.
  - byte: Very small integers, useful for small data.

Java Types: Other Types

- Floating Point (Decimal) Types:
  - float: Single-precision decimal numbers
  - double: Double-precision decimal numbers.
  - Some phone platforms do not support FP.
- String: Letters, words, or sentences.
- boolean: True or false.
- char: Single Latin Alphanumeric characters

Variable Name Rules

- Variable names (or identifiers) may be any length, but must start with:
  - A letter (a – z, A-Z),
  - A dollar sign ($),
  - Or, an underscore (_).
- Identifiers cannot contain special operation symbols like +, -, *, /, &, %, ^, etc.
- Certain reserved keywords in the Java language are illegal.
  - int, double, String, etc.

Naming Variables

- Java is case sensitive
- A rose is not a Rose is not a ROSE
- Choose variable names that are informative
  - Good: int studentExamGrade;
  - Bad: int tempvar3931;
- Camel Case”: Start variable names with lower case and capitalize each word:
  - “camelsHaveHumps”.

Review

- Which of the following are valid variable names?
  - $amount
  - 6tally
  - my*Name
  - salary
  - _score
  - first Name
  - short

Integer Types

- There are 4 primitive integer types: `byte`, `short`, `int`, `long`.
- Each type has a maximum value, based on its underlying binary representation:
  - Bytes: ± 128 (8 bits)
  - Short: ± $2^{15} \approx$ 32,000 (16 bits)
  - Int: ± $2^{31} \approx$ 2 billion (32 bits)
  - Long: ± $2^{63} \approx$ really big (64 bits)

Overflow

- What happens when if we store Bill Gates’s net worth in an int?
  - Int: ± $2^{31} \approx$ 2 billion (32 bits)
  - Bill’s net worth: > $40 billion USD

- Undefined!

Floating Point Types

- Initialize doubles as you would write a decimal number:
  - `double y = 1.23;`
  - `double w = -3.21e-10; // -3.21x10`$^{-10}$

- Doubles are more precise than Floats, but may take longer to perform operations.

Floating Point Types

- We must be careful with integer division:
  - `double z = 1/3; // z = 0.0 … Why?`

Type Casting

- When we want to convert one type to another, we use type casting
- The syntax is as follows:

```
(new type)variable
```

- Example code:
  - `double decimalNumber = 1.234;`
  - `int integerPart = (int)decimalNumber;`
- Results:
  - `decimalNumber == 1.234;`
  - `integerPart == 1;`

Boolean Type

- Boolean is a data type that can be used in situations where there are two options, either `true` or `false`.
- The values `true` or `false` are case-sensitive keywords. Not `True` or `TRUE`.
- Booleans will be used later for testing properties of data.
- Example:
  - `boolean monsterHungry = true;`
  - `boolean fileOpen = false;`

Character Type

- Character is a data type that can be used to store a single characters such as a letter, number, punctuation mark, or other symbol.

- Characters are a single letter enclosed in **single** quotes.

- Example:
  - char firstLetterOfName = 'e' ;
  - char myQuestion = '?' ;

String Type

- Strings are not a primitive. They are what’s called an Object, which we will discuss later.

- Strings are sequences of characters surrounded by **double** quotations.

- Strings have a special append operator + that creates a new String:
  - `String greeting = “Jam” + “bo”;`
  - `String bigGreeting = greeting + “!”;`

Review

- What data types would you use to store the following types of information?:

– Population of Kenya	`int`
– World Population	`long`
– Approximation of π	`double`
– Open/closed status of a file	`boolean`
– Your name	`String`
– First letter of your name	`char`
– $237.66	`double`

A Note on Statements

- A statement is a command that causes something to happen.
- All statements are terminated by semicolons ;
- Declaring a variable is a statement.
- Method (or function) calls are statements:
  - `System.out.println(“Hello, World”);`
- In lecture 4, we’ll learn how to control the execution flow of statements.

What are Operators?

- ***Expressions*** can be combinations of variables, primitives and operators that result in a value

- Operators are special symbols used for:

  - mathematical functions

  - assignment statements

  - logical comparisons

- Examples with operators:

```
3 + 5                    // uses + operator
14 + 5 – 4 * (5 – 3)     // uses +, -, * operators
```

The Operator Groups

- There are 5 different groups of operators:
  - - Arithmetic Operators
  - - Assignment Operator
  - - Increment / Decrement Operators
  - - Relational Operators
  - - Conditional Operators

- The following slides will explain the different groups in more detail.

Arithmetic Operators

- Java has the usual 5 arithmetic operators:
  - +, -, × , /, %

- Order of operations (or precedence):
  1. **P**arentheses (**B**rackets)
  2. **E**xponents (**O**rder)
  3. **M**ultiplication and **D**ivision from left to right
  4. **A**ddition and **S**ubtraction from left to right

Order of Operations (Cont'd)

- Example: `10 + 15 / 5;`

- The result is different depending on whether the addition or division is performed first

```
(10 + 15) / 5 = 5
10 + (15 / 5) = 13
```

  Without parentheses, Java will choose the second case

- You should be explicit and use parentheses to avoid confusion

Integer Division

- In the previous example, we were lucky that `(10 + 15) / 5` gives an exact integer answer (5).

- But what if we divide 63 by 35?

- Depending on the data types of the variables that store the numbers, we will get different results.

Integer Division (Cont’d)

-
  ```
  int i = 63;
  int j = 35;
  System.out.println(i / j);
  Output: 1
  ```

-
  ```
  double x = 63;
  double y = 35;
  System.out.println(x / y);
  Output: 1.8
  ```

- The result of integer division is just the integer part of the quotient!

Assignment Expression

- The basic assignment operator (=) assigns the value of `expr` to `var`

```
name = value
```

- Java allows you to combine arithmetic and assignment operators into a single statement

- Examples:

  `x = x + 5;` is equivalent to `x += 5;`

  `y = y * 7;` is equivalent to `y *= 7;`

Increment/Decrement Operators

- `++` is called the increment operator. It is used to increase the value of a variable by 1.

  For example:

  `i = i + 1;` can be written as:

  ```
  ++i; or i++;
  ```

- `--` is called the decrement operator. It is used to decrease the value of a variable by 1.

  `i = i - 1;` can be written as:

  ```
  --i; or i--;
  ```

Increment Operators (cont’d)

- The increment / decrement operator has two forms :

  - Prefix Form e.g ++i; --i;

  - Postfix Form e.g i++; i--;

Prefix increment /decrement

- The prefix form first adds/ subtracts 1 from the variable and then continues to any other operator in the expression

- Example:

```
int numOranges = 5;
int numApples = 10;
int numFruit;
numFruit = ++numOranges + numApples;

numFruit has value 16
numOranges has value 6
```

Postfix Increment/ Decrement

- The postfix form i++, i-- first evaluates the entire expression and then adds 1 to the variable
- Example:

```
int numOranges = 5;
int numApples = 10;
int numFruit;
numFruit = numOranges++ + numApples;

numFruit has value 15
numOranges has value 6
```

Relational (Comparison) Operators

- Relational operators compare two values
- They produce a boolean value (`true` or `false`) depending on the relationship

Operation	….Is true when
`a > b`	**a** is greater than **b**
`a >= b`	**a** is greater than or equal to **b**
`a == b`	**a** is equal to **b**
`a != b`	**a** is not equal to **b**
`a <= b`	**a** is less than or equal to **b**
`a < b`	**a** is less than **b**

Note: ==
sign!

Examples of Relational Operations

```
int x = 3;
int y = 5;
boolean result;
```

```
1) result = (x > y);
```

`result` is assigned the value false because 3 is not greater than 5

2) `result = (15 == x*y);`

now `result` is assigned the value true because the product of 3 and 5 equals 15

3) `result = (x != x*y);`

now `result` is assigned the value true because the product of `x` and `y` (15) is not equal to `x` (3)

Conditional Operators

Symbol	Name
&&	AND
||	OR
!	NOT

- Conditional operators can be referred to as `boolean` operators, because they are only used to combine expressions that have a value of `true` or `false`.

Truth Table for Conditional Operators

x	y	x && y	x || y	!x
True	True	True	True	False
True	False	False	True	False
False	True	False	True	True
False	False	False	False	True

Examples of Conditional Operators

```
boolean x = true;
boolean y = false;
boolean result;
```

- Let `result = (x && y);`

  `result` is assigned the value `false`

- Let `result = ((x || y) && x);`

  `(x || y)` evaluates to **`true`**
  `(true && x)` evaluates to **`true`**

  now `result` is assigned the value `true`

Using && and ||

- false && …
- true || …

- Java performs *short circuit evaluation*
  - Evaluate && and || expression s from left to right
  - Stop when you are guaranteed a value

Short-Circuit Evaluation

```
(a && (b++ > 3));
```

What happens if `a` is `false`?

- Java will not evaluate the right-hand expression `(b++ > 3)` if the left-hand operator `a` is `false`, since the result is already determined in this case to be `false`. This means b will not be incremented!

```
(x || y);
```

What happens if `x` is `true`?

- Similarly, Java will not evaluate the right-hand operator `y` if the left-hand operator `x` is `true`, since the result is already determined in this case to be `true`.

Review

1) What is the value of result?

```
int x = 8;
int y = 2;
boolean result = (15 == x * y);
```

2) What is the value of result?

```
boolean x = 7;
boolean result = (x < 8) && (x > 4);
```

3) What is the value of z?

```
int x= 5;
int y= 10;
int z= y++ + x+ ++y;
```

Appendix I: Reserved Keywords

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

Appendix II: Primitive Data Types

This table shows all primitive data types along with their sizes and formats:

Data Type	Description
`byte`	Variables of this kind can have a value from: **-128 to +127** and occupy 8 bits in memory
`short`	Variables of this kind can have a value from: **-32768 to +32767** and occupy 16 bits in memory
`int`	Variables of this kind can have a value from: **-2147483648 to +2147483647** and occupy 32 bits in memory
`long`	Variables of this kind can have a value from: **-9223372036854775808 to +9223372036854775807** and occupy 64 bits in memory

Appendix II: Primitive Data Types

Real Numbers

Data Type	Description
`float`	Variables of this kind can have a value from: **1.4e(-45) to 3.4e(+38)**
`double`	Variables of this kind can have a value from: **4.9e(-324) to 1.7e(+308)**

Other Primitive Data Types

`char`	Variables of this kind can have a value from: A single character
`boolean`	Variables of this kind can have a value from: *True* or *False*

Nuff said, time for some action!

Lab Section 2

Accelerating Information Technology Innovation

Control Structures

http://aiti.mit.edu

What are Control Structures?

- Without control structures, a computer would evaluate all instructions in a program sequentially

- Allow you to control:
  - the order in which instructions are evaluated
  - which instructions are evaluated
  - the “flow” of the program

- Use pre-established code structures:
  - block statements (anything contained within curly brackets)
  - decision statements (if, if-else, switch)
  - Loops (for, while)

Block Statements

- Statements contained within curly brackets

```
{
  statement1;
  statement2;
}
```

- Evaluated sequentially when given instruction to “enter” curly brackets
- *Most basic control structure (building block of other control structures)*

Decision Statements: if-then

The “if” decision statement causes a program to execute a statement ***conditionally****

```
    if (condition) {
        statement;
    }
 next_statement;
```

***Executes a statement when a condition is true**

Dissecting if-then

```
if (condition) {
   statement;
  }
next_statement;
```

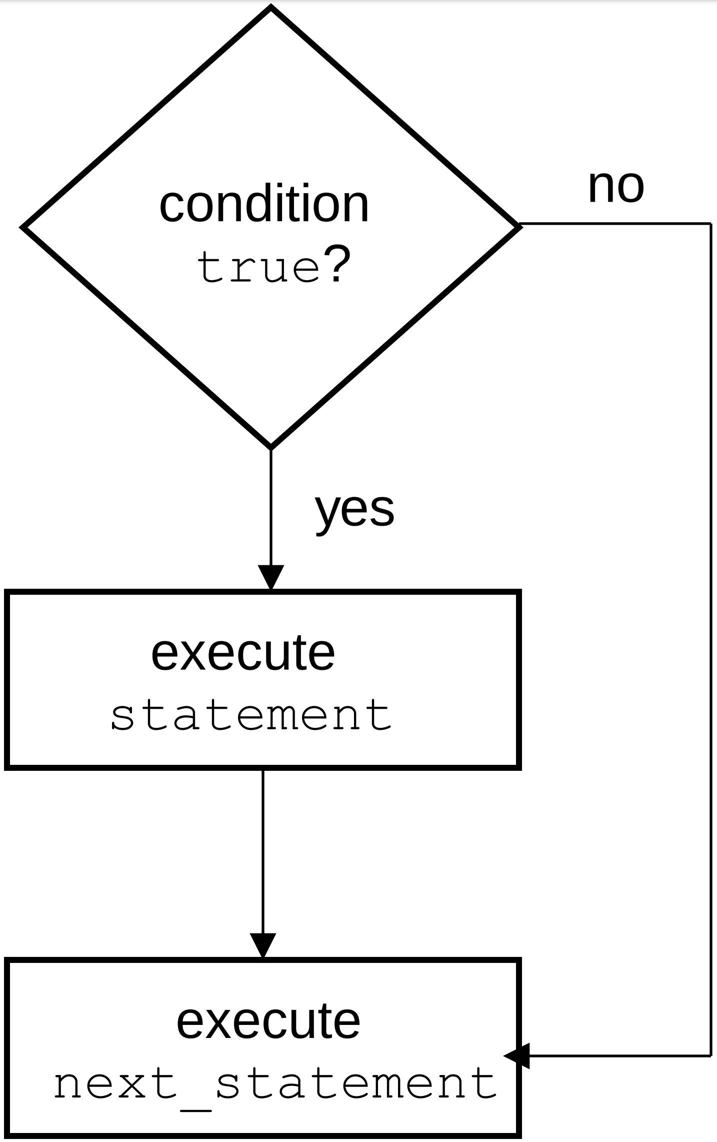
- The `condition` must produce either `true` or `false`, also known as a `boolean` value

- If `condition` returns `true`, `statement` is executed and then `next_statement`

- If `condition` returns `false`, `statement` is not executed and the program continues at `next_statement`

if-then Statement Flow Chart

```
if (condition) {
  statement;
}
next_statement;
```



if-then Example

```
int price = 5;

if (price > 3) {
  System.out.println(“Too expensive”);
}
//continue to next statement
```

Output:

```
Too expensive
```

if-then-else Statements

- The basic “if” statement can be extended by adding the “else” clause in order to do something if expression is false

```
if (condition) {
 statement1;
}
else {
 statement2;
}
next_statement;
```

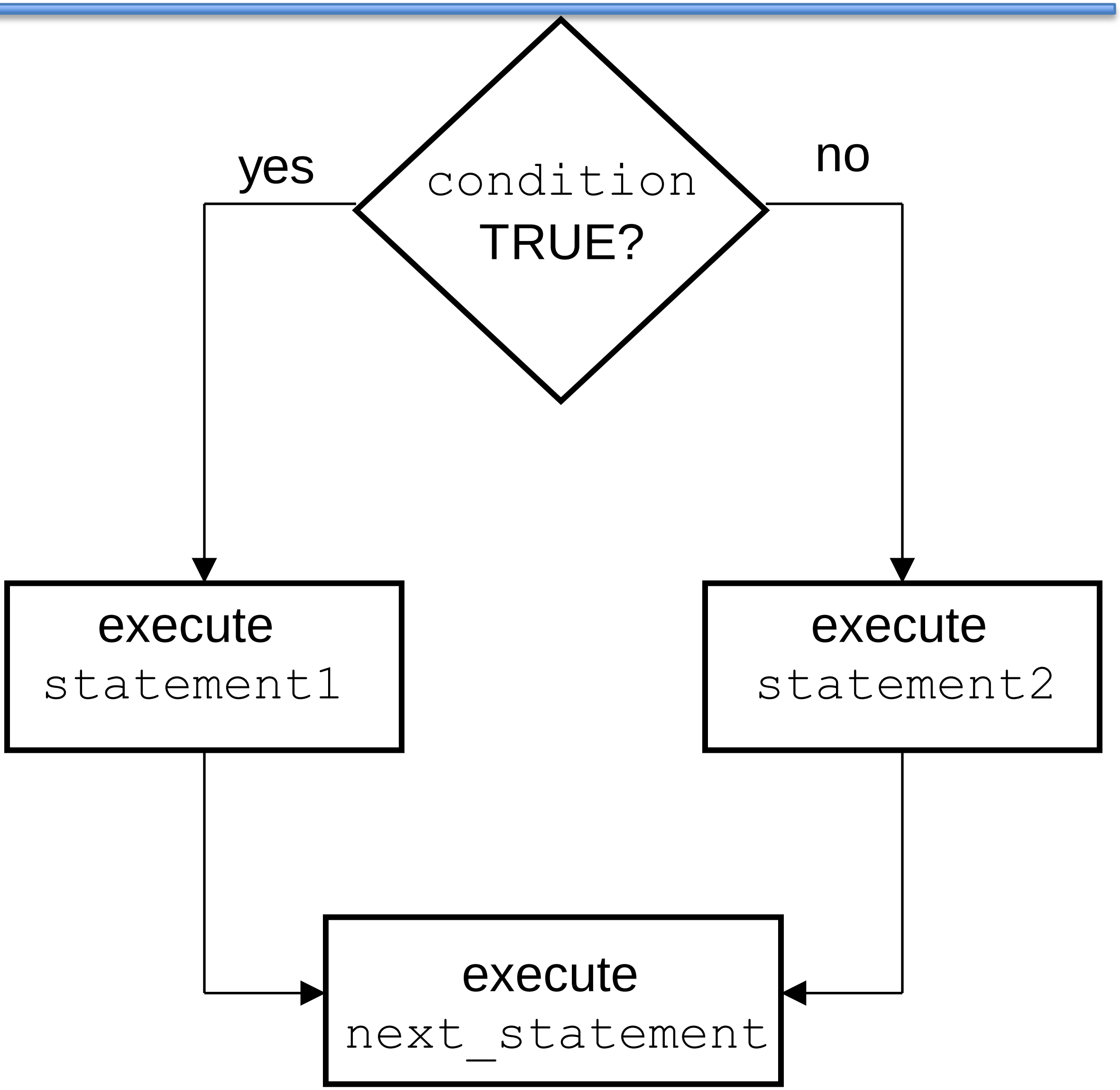
- Again, the `condition` must produce a `boolean` value

- If `condition` returns `true`, `statement1` is executed and then `next_statement` is executed.

- If `condition` returns `false`, `statement2` is executed and then `next_statement` is executed.

if-then-else Statement Flow Chart

```
if (condition){
    statement1;
}
else {
    statement2;
}
next_statement;
```



if-then-else Example

```
int price = 2;

if (price > 3) {
  System.out.println(“Too expensive”);
}
else {
  System.out.println(“Good deal”);
}
//continue to next statement
```

Output:

Good deal

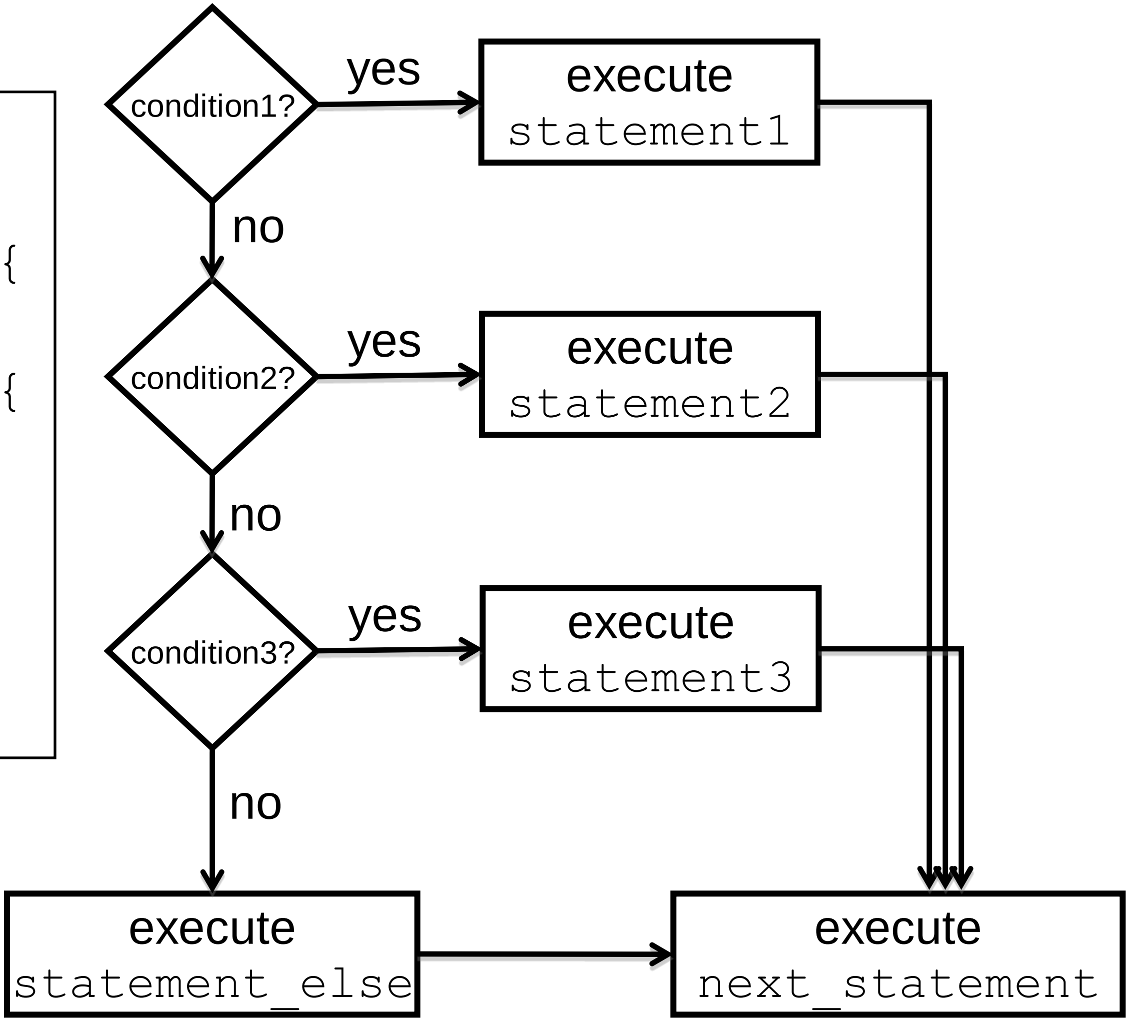
Chained if-then Statements

- Note that you can combine if-else statements below to make a chain to deal with more than one case

```
if (grade == 'A')
  System.out.println("You got an A.");
else if (grade == 'B')
  System.out.println("You got a B.");
else if (grade == 'C')
  System.out.println("You got a C.");
else
  System.out.println("You got an F.");
```

Chained if-then-else Statement Flow Chart

```
if (condition1) {
    statement1;
} else if (condition2) {
    statement2;
} else if (condition3) {
    statement3;
} else {
   statement_else;
}
next_statement;
```



switch Statements

- The `switch` statement is another way to test ***several cases*** generated by a given expression.

- The expression must produce a result of type `char`, `byte`, `short` or `int`, but not `long`, `float`, or `double`.

```
switch (expression) {

    case value1:
        statement1;
        break;

    case value2:
        statement2;
        break;

    default:
        default_statement;
        break;
}
```

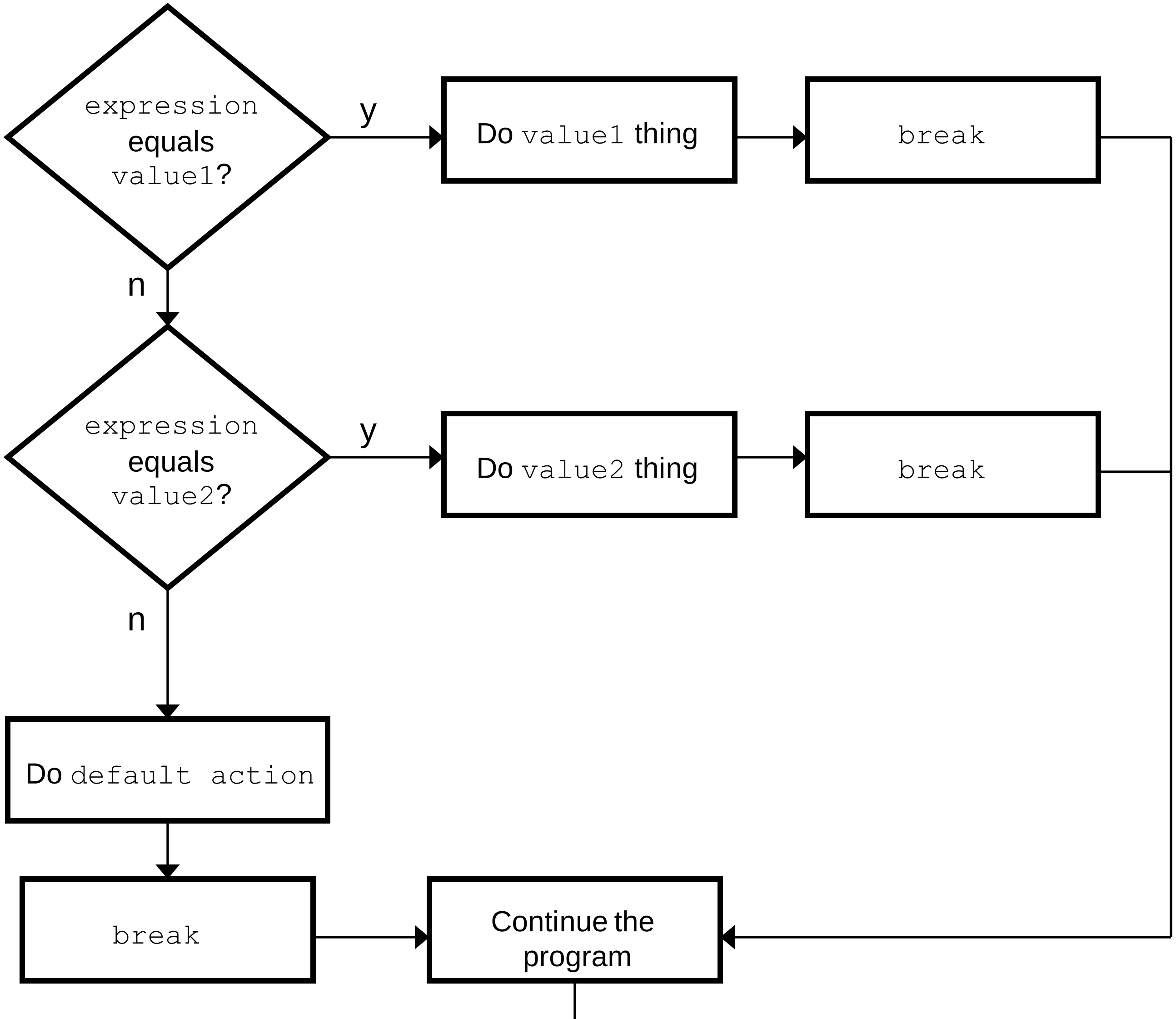
- The `break;` statement exits the switch statement

switch Statement Flow Chart

```
switch (expression){
    case value1:
     // Do value1 thing
     break;

    case value2:
     // Do value2 thing
     break;

    ...
    default:
     // Do default action
     break;
}
// Continue the program
```



Remember the Example…

- Here is the example of chained if-else statements:

```
if (grade == 'A')
 System.out.println("You got an A.");

else if (grade == 'B')
 System.out.println("You got a B.");

else if (grade == 'C')
 System.out.println("You got a C.");

else
 System.out.println("You got an F.");
```

Chained if-then-else as switch

- Here is the previous example as a switch

```
switch (grade) {
   case 'A':
      System.out.println("You got an A.");
      break;
   case 'B':
      System.out.println("You got a B.");
      break;
   case 'C':
      System.out.println("You got a C.");
      break;
   default:
      System.out.println("You got an F.");
}
```

What if there are no breaks?

- Without break, switch statements will execute the first statement for which the expression matches the case value AND then evaluate all other statements from that point on

- For example:

```
switch (expression) {

    case value1:
        statement1;

    case value2:
        statement2;

    default:
        default_statement;
}
```

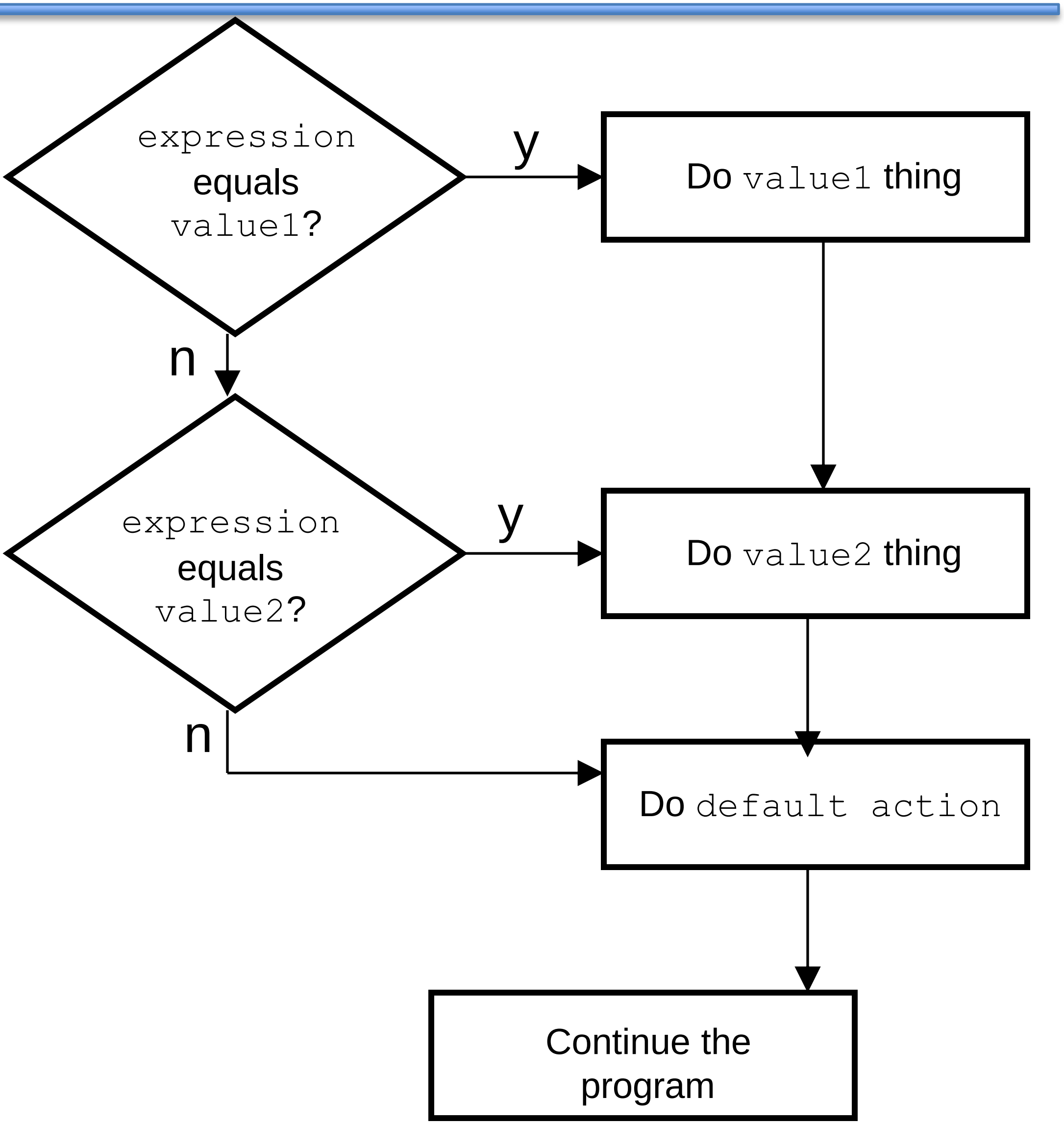
- NOTE: **Every statement after the true case is executed**

Switch Statement Flow Chart w/o breaks

```
switch (expression){
    case value1:
     // Do value1 thing

    case value2:
     // Do value2 thing

    ...
    default:
     // Do default action
}
// Continue the program
```



Loops

- A loop allows you to execute a statement or block of statements repeatedly.

- There are 4 types of loops in Java:

1. `while` loops
2. `do-while` loops
3. `for` loops
4. `foreach` loops (coming soon!)

The `while` Loop

```
while (condition){
   statement
}
```

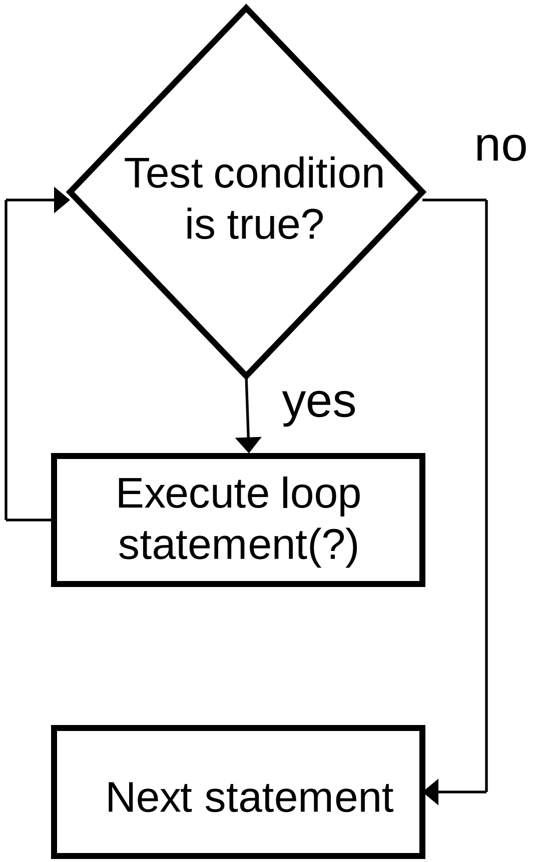
- This while loop executes as long as `condition` is `true`. When `condition` is `false`, execution continues with the statement following the loop block.

- The condition is tested at the beginning of the loop, so if it is initially `false`, the loop will not be executed at all.

while Loop Flow Chart

The while loop

```
while (expression){
    statement
}
```



Example

```
 int limit = 4;
 int sum = 0;
 int i = 1;

while (i < limit){
   sum += i;
         i++;
}
```

i = 1	sum = 1
i = 2	sum = 3
i = 3	sum = 6
i = 4	

- What is the value of `sum` ?

6

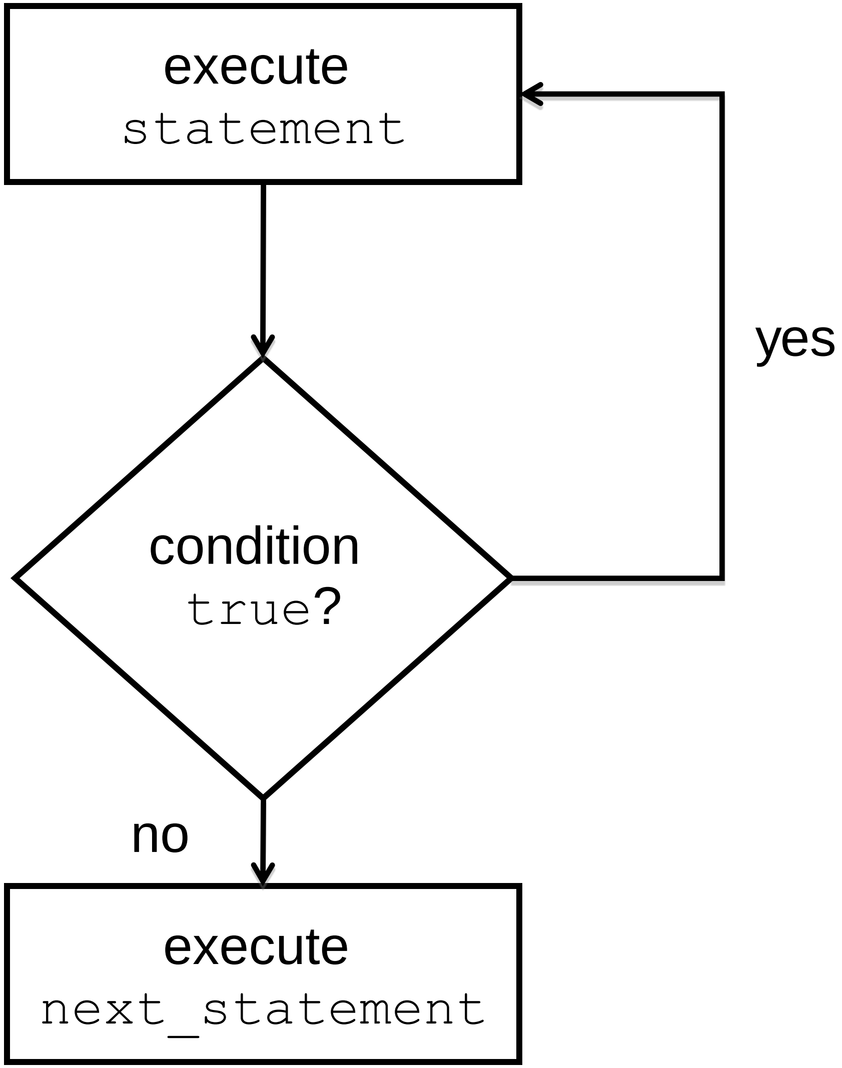
do-while Loops

- Similar to while loop but guarantees **at least one** execution of the body

```
do {
  statement;
}
while(condition
)
```

do-while Flowchart

```
do {
  statement;
}
while(condition)
next_statement;
```



do-while Example

```
boolean test = false;


do {
  System.out.println(“Hey!”)
}
while(test)
```

Output:

```
Hey!
```

for Loop

- Control structure for capturing the most common type of loop

```
i = start;
while (i <= end)
{
   . . .
   i++;
}
```

⇔

```
for (i = start; i <= end; i++)
{
      ...
}
```

Dissecting the for Loop

```
for (initialization; condition; update)
{
     statement;
}
```

The control of the `for` loop appear in parentheses and is made up of three parts.

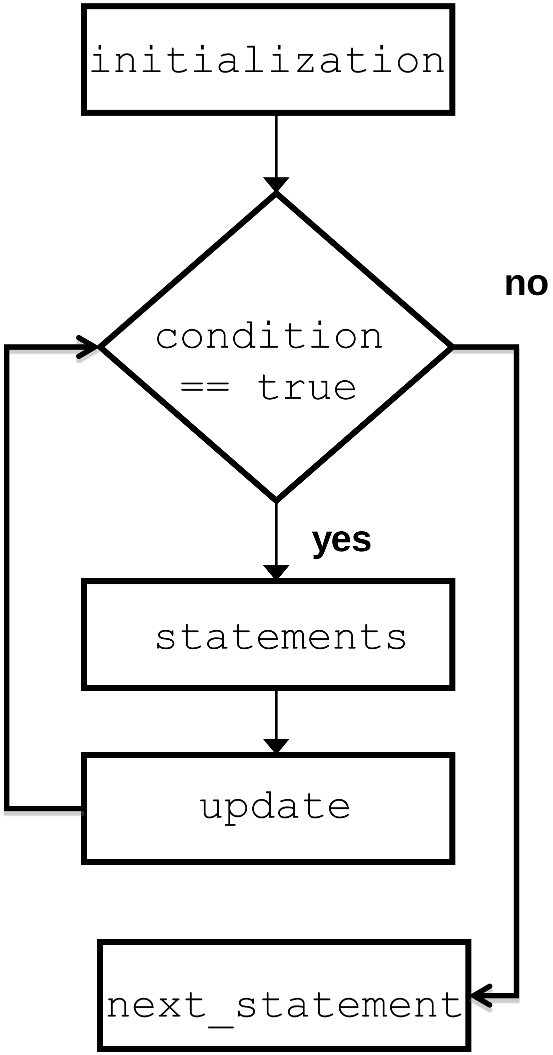
1. The first part, the `initialization`, sets the initial conditions for the loop and is executed before the loop starts.

2. Loop executes so long as the `condition` is true and exits otherwise

1. The third part of the control information, the `update`, is used to increment the loop counter. This is executed at the end of each loop iteration.

for Loop Flow Chart

The `for` loop



```
for (initialization;
     condition;
     update)
{
    //statements
}
next_statement;
```

Example

```
int limit = 4;
int sum = 0;

for(int i = 1; i<=limit; i++ )
{
  sum += i;
}
```

i = 1	sum = 1
i = 2	sum = 3
i = 3	sum = 6
i = 4	sum = 10
i = 5	-- --

- What is the value of `sum` ?

10

Another Example

```
for ( int div = 0; div<1000; div++ ) {

 if ( div % 12 == 0 ){

    System.out.println(div+"is divisible by 12");

 }
}
```

- This loop will display every number from 0 to 999 that is evenly divisible by 12.

Other Possibilities

- If there is more than one variable to set up or increment they are separated by a comma.

```
for (i=0, j=0; i*j<1000; i++, j+=2) {

   System.out.println(i+"*"+j+"="+i*j);


}
```

- You do not have to fill every part of the control of the `for` loop but you must still have two semi-colons.

```
for (int i=0; i<100; ) {

   sum+=i;
       i++;

}
```

*Straying far from convention may make code difficult to understand and thus is **not common**

Using the break Statement in Loops

- We have seen the use of the break statement in the switch statement.
- In loops, you can use the break statement to exit the current loop you are in. Here is an example:

```
int index = 0;
while (index <= 4) {
 index++;
 if (index == 3)
    break;
   System.out.println("The index is “
          + index);
}
```

```
index = 1     The index is 1
index = 2     The index is 2
index = 3
```

Using the continue Statement in Loops

- Continue statement causes the loop to jump to the next iteration
- Similar to break, but only skips to next iteration; doesn't exit loop completely

```
int index = 0;
while (index <= 4){
 index++;
 if (index == 3)
    continue;
   System.out.println("The index is “
      + index);
}
```

index = 1	The index is 1
index = 2	The index is 2
index = 3	-- --
Index = 4	The index is 4

Nested Loops – Example

- Printing a triangle

```
for (int i=1; i<=5; i++){
   for (int j=1; j<=i; j++){
      System.out.println(“*”);
   }
}
```

```
*
**
***
****
*****
```

Control Structures Review Questions

You are withdrawing money from a savings account.

How do you use an If Statement to make sure you do not withdraw more than you have?

```
if ( amount < balance )
 {
    balance = balance - amount;
}
//next statement
```

Which Control Structure?

- As a programmer, you will never be asked something like: “Write a for loop to…”
- You will need to implement logic in your program that meets your specification and requirements
- With experience, you will know which control structure to use.

Play time!

Lab Section 3

MIT Accelerating Information Technology Innovation

Arrays

What are Arrays?

- An array is a series of compartments to store data.
- Essentially a block of variables.
- In Java, arrays can only hold one type.
- For example, **`int`** arrays can hold only integers and **`char`** arrays can only hold characters.

Array Visualization and Terms

- Arrays have a type, name, and size.
- Array of three integers named `prices` :
  - `prices` :

int	int	int

- Array of four Strings named `people`:
  - `people` :

	String	String	String	String
(Indices)	0	1	2	3

- We refer to each item in an array as an *element*.
- The position of each element is known as its *index.*

Declaring an Array

- Array declarations similar to variables, but use square brackets:
  - `datatype[] name;`
- For example:
  - `int[] prices;`
  - `String[] people;`
- Can alternatively use the form:
  - `datatype name[];`
  - `int prices[];`

Allocating an Array

- Unlike variables, we need to *allocate* memory to store arrays. (*malloc()* in C.)
- Use the `new` keyword to allocate memory:
  - `name = new type[size];`
  - `prices = new int[3];`
  - `people = new String[5];`
- This allocates an integer array of size 3 and a String array of size 5.
- Can combine declaration and allocation:
  - `int[] prices = new int[3];`

Array Indices

- Every element in an array is referenced by its index.
- In Java, the index starts at 0 and ends at *n-1*, where *n* is the size of the array.
- If the array `prices` has size 3, its valid indices are 0, 1, and 2.
- Beware “Array out of Bounds” errors.

Using an Array

- We access an element of an array using square brackets `[]`:
  - `name[index]`
- Treat array elements just like a variable.
- Example assigning values to each element of `prices`:
  - `prices[0] = 6;`
  - `prices[1] = 80;`
  - `prices[2] = 10;`

Using an Array

- We assign values to elements of String arrays in a similar fashion:
  - String[] people;
  - people = new String[5];
  - people[0] = ”Michael”;
  - people[1] = ”Michelle”;
  - people[2] = ”Cory”;
  - people[3] = ”Zach”;
  - people[4] = ”Julian”;

Initializing Arrays

- You can also specify all of the items in an array at its creation.
- Use curly brackets to surround the array’s data and separate the values with commas:

  -
    ```
    String[] people = {“Michael”,
     “Michelle”, “Zach”, “Cory”,
    “Julian”};
    ```

  -
    ```
    int[] prices = {6, 80, 10};
    ```
- All the items must be of the same type.

Vocabulary Review

- Allocate - Create empty space that will contain the array.
- Initialize - Fill in a newly allocated array with initial values.
- Element - An item in the array.
- Index - Element’s position in the array.
- Size or Length - Number of elements.

Review 1

Which of the following sequences of statements does not create a new array?

a) `int[] arr = new int[4];`

b)
```
int[] arr;
arr = new int[4];
```

c) `int[] arr = { 1, 2, 3, 4};`

d) `int[] arr;`

Lengths of Array

- Each array has a default *field* called `length`
- Access an array’s `length` using the format:
  - `arrayName.length;`
- Example:

  -
    ```
    String[] people = {“Michael”,
     “Michelle”, “Zachary”, “Cory”, “Julian”};
    ```
  - `int numPeople = people.length;`
- The value of `numPeople` is now 5.

- Arrays are always of the same size. Their lengths cannot be changed once they are created.

Example

- Sample Code:

```
String[] people = {“Gleb”,
  “Lawrence”, “Michael”,
  “Stephanie”, “Zawadi”};
for(int i=0; i<names.length; i++)
  System.out.println(names[i]+”!");
```

- Output:

```
Gleb!
Lawrence!
Michael!
Stephanie!
Zawadi!
```

Review

- Given this code fragment:
  - `int[] data = new int[10];`
  - `System.out.println(data[j]);`
- Which are legal values of `j`?

a) -1

b) 0

c) 3.5

d) 10

Review

- Decide what type and size of array (if any) to store each data set:
  - Score in each quarter of a football game.

```
int[] quarterScore = new int[4];
```

  - Your name, date of birth, and height.

```
Not appropriate. Different types.
```

  - Hourly temperature readings for a week.

```
float[] tempReadings = new float[168];
```

  - Your daily expenses for a year.

```
float[] dailyExpenses = new float[365];
```

Exercise

- What are the contents of `c` after the following code segment?

```
int [] a = {1, 2, 3, 4, 5};
int [] b = {11, 12, 13};
int [] c = new int[4];
for (int j = 0; j < 3; j++) {
   c[j] = a[j] + b[j];
}
```

2-Dimensional Arrays

- The arrays we've used so far can be thought of as a single row of values.
- A 2-dimensional array can be thought of as a grid (or matrix) of values.
- Each element of the 2-D array is accessed by providing two indices: a row index and a column index.
- A 2-D array is actually just an array of arrays

	0	1
0	8	4
1	9	7
2	3	6

value at row index 2,
column index 0 is 3

2-D Array Example

- Example: A landscape grid of a 20 x 55 acre piece of land. We want to store the height of the land at each row and each column of the grid.
- We declare a 2-D array two sets of square brackets:
  - `double[][] heights;`
  - `heights = new double[20][55];`
- This 2-D array has 20 rows and 55 columns
- To access the acre at row index 11 and column index 23: `heights[11][23]`

Lights, Camera, Action!

Lab Section 4

MIT Accelerating Information Technology Innovation

Methods

Agenda

- What a method is
- Why we use methods
- How to declare a method
- The four parts of a method
- How to use (invoke) a method
- The purpose of the main method

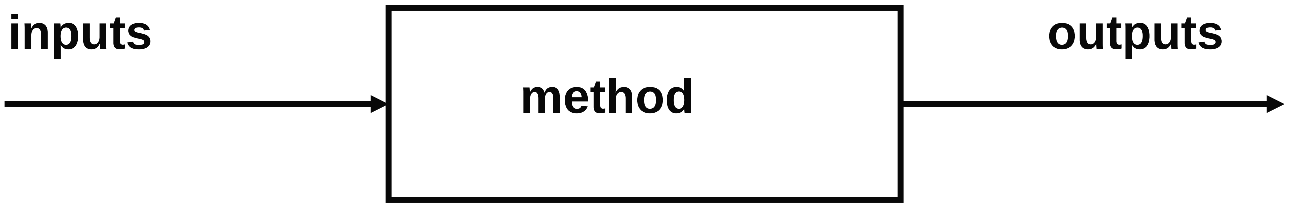
The Concept of a Method

- Methods are a way of organizing a sequence of statements into a named unit.
  - Reusable
  - Parameterizable (can accept inputs)
  - Organize code into smaller units
    - Easier to understand

- Any complex process that can exist on its own should be a method
  - Better to have more methods, even if they are not reused.

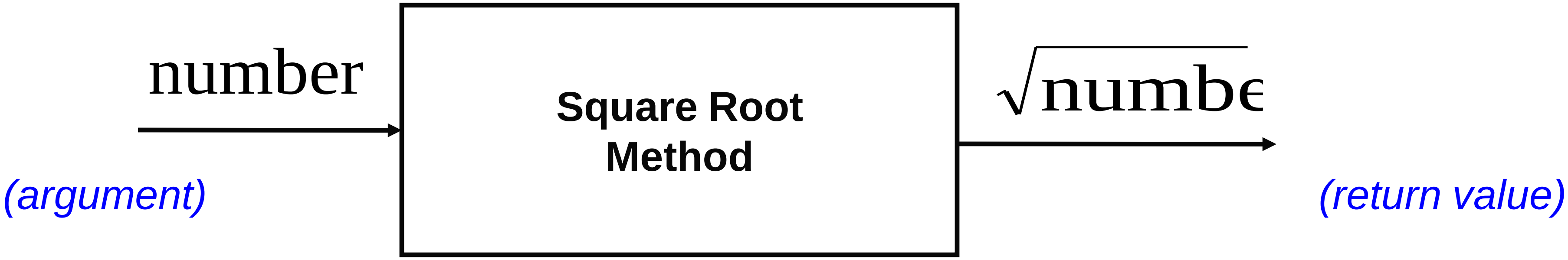
The Concept of a Method

- Methods can accept inputs (called arguments)
- They can then perform some operations with the arguments
- And can output a value (called a return value) that is the result of the computations



Square Root Method

- The square root method accepts a single number as an argument and returns the square root of that number.



number

(argument)

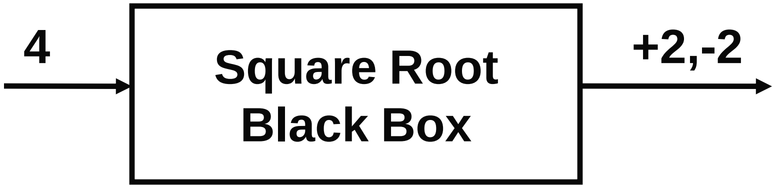
Square Root Method

$\sqrt{\text{numbe}}$

(return value)

Square Root Method (con’t)

- The computation of square roots involves many intermediate steps between input and output.
- When we use square root, we don’t care about these steps or details. All we need is to get the correct output.
- Hiding the internal workings of a method and providing the correct answer is known as *abstraction*



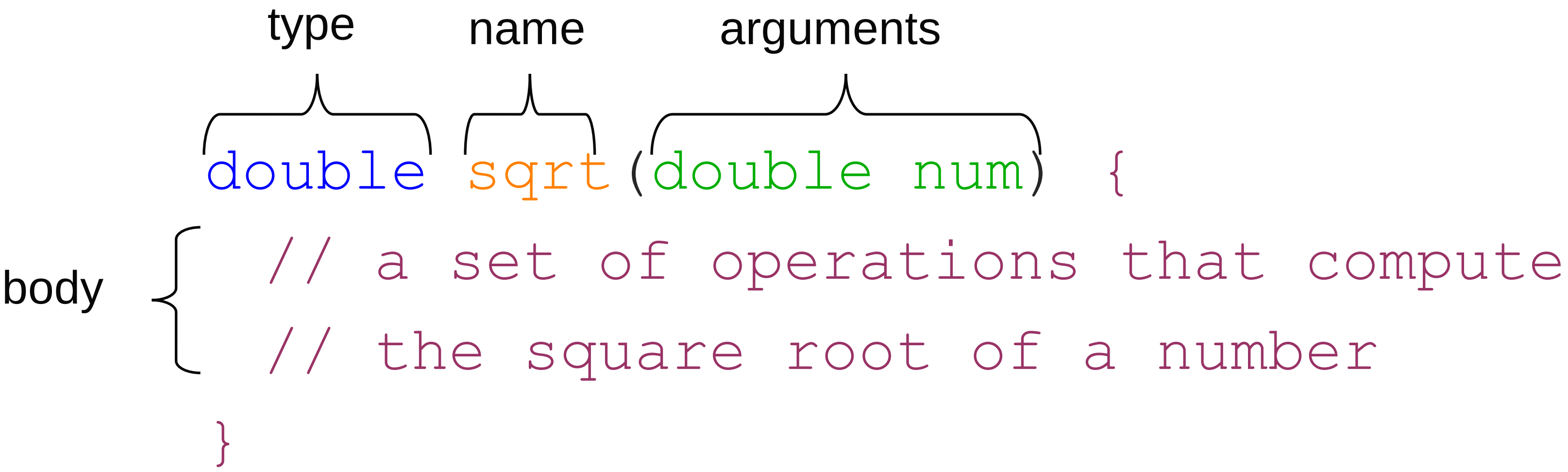
4

Square Root Black Box

+2,-2

Declaring Methods

- A method has 4 parts: the return type, the name, the arguments, and the body:



type name arguments

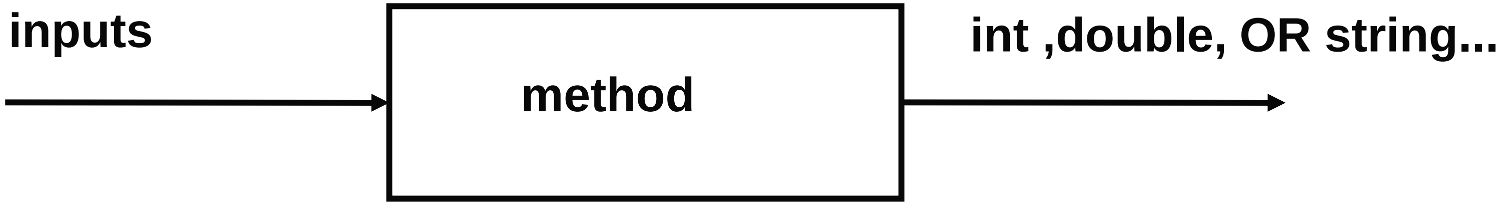
```
double sqrt(double num)  {
  // a set of operations that compute
  // the square root of a number
}
```

body

- The type, name and arguments together is referred to as the *signature* of the method
- Methods with same names must have unique signature

Return Type of a Method

- The return type of a method may be any data type....



- The return type of a method is a promise for what data type the output will be
  - A method can return different outputs than inputs
  - A method cannot return multiple types, returns one type
- Methods can also return nothing in which case they are declared void.

Return Statements

- The return statement is used in a method to output the result of the method computation.
- It has the form:
    - `return expression-value;`
- The type of the expression_value must be the same as the type of the method:

```
double sqrt(double num) {
    double answer;
    // Compute the square root of num
    // and store in answer
    return answer;
}
```

- What is the return type of this method?

Return Statements

A method exits immediately after it executes the return statement

```
double sqrt(double num) {
  double answer;
  // Compute the square root of num
  // and store in answer
  return answer;

  answer = 5 + 4; //never executed, illegal
}
```

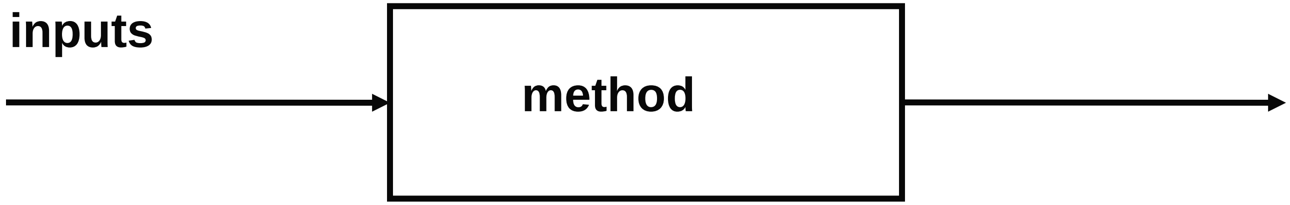
Multiple Returns

- An example using multiple returns:

```
int absoluteValue (int num) {
 if (num < 0)
   return -num;
 else
   return num;
}
```

void Methods

- A method of type **`void`** does not return a value



- Used often in practice.
  - Perform some computation that does not produce a value
  - Affect system state, ex: System.out.println()
- A void method can have a return statement without any specified value. i.e. **`return;`**
- If no return statement is used in a method of type void, it automatically returns at the end

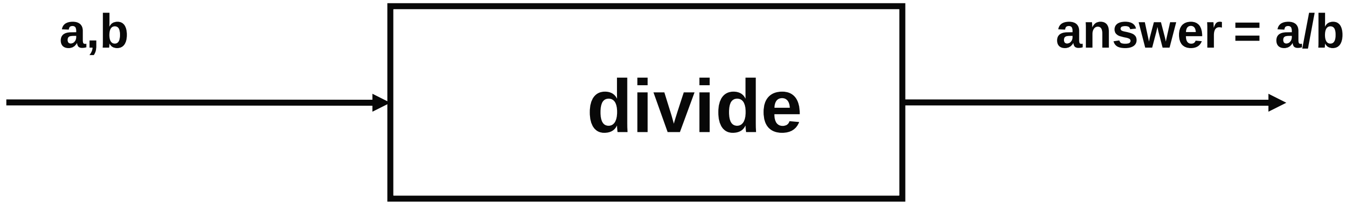
Method Arguments

- Methods can take input in the form of arguments.
- Arguments are used as variables inside the method body.
- Like variables, arguments must have their type specified.
- Arguments are specified inside the parentheses that follow the name of the method.

Example Method

- Here is an example of a method that divides two doubles:

```
double divide(double a, double b) {
  double answer;
  answer = a / b;
  return answer;
}
```



Method Arguments

- Multiple method arguments are separated by commas:

```
double divide(double a, double b) {
   double answer;
   answer = a / b;
   return answer;
}
```

- Arguments may be of different types (double/int)
  - `double divide(int a, int b)`
- When calling method, exact sequence of input types must be applied

The Method Body

- The body of a method is a block specified by curly brackets i.e { }. The body defines the actions of the method.

- The method arguments can be used anywhere inside of the body.

- All methods must have curly brackets to specify the body even if the body contains only one statement or no statements.

```
double divide(
 double a, double b)
 {
    double answer;
    answer = a / b;
    return answer;
 }
```

Invoking Methods

- To call a method, specify the name of the method followed by a list of comma separated arguments in parentheses:

```
divide(10, 2); //Computes 10/2
```

- If the method has no arguments, you still need to follow the method name with empty parentheses:

```
      int size() {
        //Compute and return size
      }
     …

   size(); //Calls size
```

Method Variable Scoping

- For now, methods can only access their own arguments and local variables.
  - A method cannot access arguments/locals from other methods
  - Even if one method calls another

- Example…

Recursive Methods

- A method can also call itself!
  - When a method calls itself, it needs a stopping condition, called the base case
    - Or else it would call itself without end
  - Example Factorial:
- Factorial of n, denoted n!:
  - n × (n – 1) × (n – 2) × … × 0
  - 0! = 1 (base case)

Factorial Implementation

```
int factorial(int n) {

  if (n==0)
   return 1;
  else {
   return n *
       factorial (n-1);
  }
}
```

Static Methods

- For now, all the methods we write in lab will be static.

```
static double divide(double a,
                     double b) {
   return a / b;
}
```

- We'll learn what it means for a method to be static in a later lecture

main – A Special Method

- The only method that we have used in lab up until this point is the **`main`** method.
- The main method is where a Java program always starts when you run a class file (entry point)
- The **`main`** method is static and has a strict signature which must be followed:

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

main Method (con’t)

```
class SayHi {
   public static void main(String[] args) {
     System.out.println("Hi, " + args[0]);
  }
}
```

- If you were to type `java Program arg1 arg2` … `argN` on the command line, anything after the name of the class file is automatically entered into the `args` array:

```
java SayHi Sonia
```

- In this example `args[0]` will contain the String "Sonia", and the output of the program will be "Hi, Sonia".

Methods Review

- What are the four parts of a method and what are their functions?

  1. **Return type** – data type returned by the method
  2. **Name** – name of the method
  3. **Arguments** – inputs to the method
  4. **Body** – sequence of instructions executed by the method

What is wrong with the following?

```
static double addSometimes(num1, num2){
   double sum;
   if (num1 < num2){
    sum = num1 + num2;
    String completed = “completed”;
    return completed;
   }
}
```

- Types for the arguments num1 and num2 are not specified
- String completed does not match the correct double return type
- Method addSometimes does not always return an answer. This will cause an error in Java because we specified that addSometimes would always return a double.

Example

```
class Max {
    public static void main(String args[]) {
        if (args.length == 0) return;

        int max = Integer.parseInt(args[0]);
        for (int i=1; i < args.length; i++) {
            if (Integer.parseInt(args[i]) > max) {
                max = Integer.parseInt(args[i]);
     }
    }
    System.out.println(max);
    }
}
```

After compiling, if you type `java Max 3 2 9 2 4` the program will print out `9`

Important Points Covered ….

- Methods capture a piece of computation we wish to perform repeatedly into a single abstraction
- Methods in Java have 4 parts: return type, name, arguments, body.
- The return type and arguments may be either primitive data types (i.e. int) or complex data types (i.e. Objects), which we will cover next lecture
- **main** is a special Java method which the java interpreter looks for when you try to run a class file
- **main** has a strict signature that must be followed:

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

Let’s get to work!

Lab Section 5