Day 1

COMP 1006/1406A

Introduction to Computer Science II

M. Jason Hinek
Carleton University

Today's agenda

- Java Basics
  - hello world
  - compiling and running a Java program
  - variables (primitive data types and objects)
  - operators
  - expressions, statements, blocks
  - control flow
  - command line arguments
  - functions
- Assignment 1

announcements

The Paul Menton Centre for Students with Disabilities is looking for a Volunteer Notetaker for this class.

The criteria for being a Volunteer Notetaker includes:

- Attending all (most) assigned classes
- Taking complete and legible notes
- typing up lecture notes and submitting them electronically within 48-hours after each Class (math & other formula heavy notes can be scanned at the Notetaking Office)
- Maintaining confidentiality

announcements

The Paul Menton Centre for Students with Disabilities is looking for a Volunteer Notetaker for this class.

What do you get out of this?

- You get to help out fellow students in the class
- Making good notes will help you learn and retain the information
- A good addition to your resume
- Can count towards your Co-Curricular Record

See `http://www1.carleton.ca/pmc/faculty/guides/notetaking/` for more information

Getting started with Java

Getting started with Java

Java is an object oriented language that is based on **classes** and **objects**.

A **class** creates a new data type that has both state and behaviour.
An **object** is an instantiation of a class (having both state and behaviour).

- state
  - have attributes (or fields) that store data
  - should have seen this in COMP 1005/1405
- behaviour
  - have methods (functions) that are part of the classes/objects
  - methods typically act on (use and modify) the state of an object
  - we will spend a lot of time with this in COMP 1006/1406

Note: a class is like a cookie cutter and objects are the cookies it makes

Hello, world!

```
# Python
print("hello, world!")
```

```
/* Java hello world */
public class HelloWorld{
  public static void main(String[] args){
    System.out.println("hello, world!");
  }
}
```

Hello, world!

```
# Python
print("hello, world!")
```

```
/* Java hello world */
public class HelloWorld{
  public static void main(String[] args){
    System.out.println("hello, world!");
  }
}
```

Hello, world!

```
# Python
print("hello, world!")
```

```
/* Java hello world */
public class HelloWorld{
  public static void main(String[] args){
    System.out.println("hello, world!");
  }
}
```

Deconstructing hello world

```
/* Java hello world */
public class HelloWorld{
   public static void main(String[] args){
    System.out.println("hello, world!");
  }
}
```

- declares/defines a class called `HelloWorld`
- declares/defines a method in the class called `main`
  - inputs an array of `String` objects
  - return type is `void` (it returns nothing)
- calls the function `println` with input string `"hello, world!"`
  - `System` is a class with three attributes (`out`, `in` and `err`)
  - `out` is the "standard" output stream
  - `out` is an instance of the `PrintStream` class

Deconstructing hello world

```java
/* Java hello world */
public class HelloWorld{
  public static void main(String[] args){
    System.out.println("hello, world!");
  }
}
```

- declares/defines a class called `HelloWorld`
- declares/defines a method in the class called `main`
  - inputs an array of `String` objects
  - return type is `void` (it returns nothing)
- calls the function `println` with input string `"hello, world!"`
  - `System` is a class with three attributes (`out`, `in` and `err`)
  - `out` is the "standard" output stream
  - `out` is an instance of the `PrintStream` class

Deconstructing hello world

```
/* Java hello world */
public class HelloWorld{
  public static void main(String[] args){
    System.out.println("hello, world!");
  }
}
```

- declares/defines a class called HelloWorld
- declares/defines a method in the class called main
  - inputs an array of String objects
  - return type is void (it returns nothing)
- calls the function println with input string "hello, world!"
  - System is a class with three attributes (out, in and err)
  - out is the "standard" output stream
  - out is an instance of the PrintStream class
  - out is a PrintStream object

Deconstructing hello world

```
/* Java hello world */
public class HelloWorld{
  public static void main(String[] args){
    System.out.println("hello, world!");
  }
}
```

- declares/defines a class called `HelloWorld`
- declares/defines a method in the class called `main`
  - inputs an array of `String` objects
  - return type is `void` (it returns nothing)
- calls the function `println` with input string `"hello, world!"`
  - `System` is a class with three attributes (`out`, `in` and `err`)
  - `out` is the "standard" output stream
  - `out` is an instance of the `PrintStream` class
  - `out` is a `PrintStream` object

Deconstructing hello world

```
/* Java hello world */
public class HelloWorld{
   public static void main(String[] args){
     System.out.println("hello, world!");
   }
}
```

- declares/defines a class called `HelloWorld`
- declares/defines a method in the class called `main`
  - inputs an array of `String` objects
  - return type is `void` (it returns nothing)
- calls the function `println` with input string `"hello, world!"`
  - `System` is a class with three attributes (`out`, `in` and `err`)
  - `out` is the "standard" output stream
  - `out` is an instance of the `PrintStream` class
  - `out` is a `PrintStream` object

Deconstructing hello world

```
/* Java hello world */
public class HelloWorld{
  public static void main(String[] args){
   System.out.println("hello, world!");
  }
}
```

- declares/defines a class called `HelloWorld`
- declares/defines a method in the class called `main`
  - inputs an array of `String` objects
  - return type is `void` (it returns nothing)
- calls the function `println` with input string `"hello, world!"`
  - `System` is a class with three attributes (`out`, `in` and `err`)
  - `out` is the "standard" output stream
  - `out` is an instance of the `PrintStream` class
  - `out` is a `PrintStream` object

Deconstructing hello world (again)

```
/* Java hello world */
public class HelloWorld{
  public static void main(String[] args){
    System.out.println("hello, world!");
  }
}
```

- public here is a top level access modifier
  - it specifies the access level of the class
  - it is either public or left empty (sometimes called friendly)
- public here is a member level access modifier
  - it specifies the access level of an attribute or a method
  - it is either public, private, protected, or left empty (friendly)

Deconstructing hello world (again)

```
/* Java hello world */
public class HelloWorld{
  public static void main(String[] args){
    System.out.println("hello, world!");
  }
}
```

- public here is a top level access modifier
  - it specifies the access level of the class
  - it is either public or left empty (sometimes called friendly)
- public here is a member level access modifier
  - it specifies the access level of an attribute or a method
  - it is either public, private, protected, or left empty (friendly)

Deconstructing hello world (again)

```
/* Java hello world */
public class HelloWorld{
  public static void main(String[] args){
    System.out.println("hello, world!");
  }
}
```

- `public` here is a top level access modifier
  - it specifies the access level of the class
  - it is either `public` or left empty (sometimes called `friendly`)
- `public` here is a member level access modifier
  - it specifies the access level of an attribute or a method
  - it is either `public`, `private`, `protected`, or left empty (`friendly`)

Deconstructing hello world (again)

```
/* Java hello world */
public class HelloWorld{
  public static void main(String[] args){
    System.out.println("hello, world!");
  }
}
```

- `public` here is a top level access modifier
  - it specifies the access level of the class
  - it is either `public` or left empty (sometimes called `friendly`)
- `public` here is a member level access modifier
  - it specifies the access level of an attribute or a method
  - it is either `public`, `private`, `protected`, or left empty (`friendly`)

Deconstructing hello world (again)

```
/* Java hello world */
public class HelloWorld{
  public static void main(String[] args){
    System.out.println("hello, world!");
  }
}
```

- `static` is a (non access) modifier
  - declares `main` to be a **class method** (it belongs to the class not to individual objects)
  - it allows the `main` method to be called without needing an instance of the `HelloWorld` class
  - there are several different modifiers (for attributes and methods)

Deconstructing hello world (again)

```
/* Java hello world */
public class HelloWorld{
  public static void main(String[] args){
    System.out.println("hello, world!");
  }
}
```

- `static` is a (non access) modifier
  - declares `main` to be a **class method** (it belongs to the class not to individual objects)
  - it allows the `main` method to be called without needing an instance of the `HelloWorld` class
  - there are several different modifiers (for attributes and methods)

Running hello world

```
/* Java hello world */
public class HelloWorld{
  public static void main(String[] args){
    System.out.println("hello, world!");
  }
}
```

- Java convention is that
  - class name is capitalized (use camel case if more than one word)
  - class `XXX` must be in the file `XXX.java`
  - so `HellowWorld` must be in the file `HelloWorld.java`
- first we need to compile the source code into Java bytecode
  - IDE will have a compile button
  - `javac HelloWorld.java` from console window (shell)
  - this creates `HelloWorld.class`, which is the Java bytecode
- next, we run the bytecode with the JVM (Java virtual machine)
  - `java HelloWorld` from the console window runs out program!
  - the JVM executes the `main` method of our program

Running hello world

```
/* Java hello world */
public class HelloWorld {
  public static void main(String[] args){
    System.out.println("hello, world!");
  }
}
```

- Java convention is that
  - class name is capitalized (use camel case if more than one word)
  - class `XXX` must be in the file `XXX.java`
  - so `HellowWorld` must be in the file `HelloWorld.java`
- first we need to compile the source code into Java bytecode
  - IDE will have a compile button
  - `javac HelloWorld.java` from console window (shell)
  - this creates `HelloWorld.class`, which is the Java bytecode
- next, we run the bytecode with the JVM (Java virtual machine)
  - `java HelloWorld` from the console window runs out program!
  - the JVM executes the `main` method of our program

Running hello world

```
/* Java hello world */
public class HelloWorld{
  public static void main(String[] args){
    System.out.println("hello, world!");
  }
}
```

- Java convention is that
  - class name is capitalized (use camel case if more than one word)
  - class `XXX` must be in the file `XXX.java`
  - so `HellowWorld` must be in the file `HelloWorld.java`
- first we need to compile the source code into Java bytecode
  - IDE will have a compile button
  - `javac HelloWorld.java` from console window (shell)
  - this creates `HelloWorld.class`, which is the Java bytecode
- next, we run the bytecode with the JVM (Java virtual machine)
  - `java HelloWorld` from the console window runs out program!
  - the JVM executes the `main` method of our program

Running hello world

```
/* Java hello world */
public class HelloWorld{
  public static void main(String[] args){
    System.out.println("hello, world!");
  }
}
```

- Java convention is that
  - class name is capitalized (use camel case if more than one word)
  - class `XXX` must be in the file `XXX.java`
  - so `HellowWorld` must be in the file `HelloWorld.java`
- first we need to compile the source code into Java bytecode
  - IDE will have a compile button
  - `javac HelloWorld.java` from console window (shell)
  - this creates `HelloWorld.class`, which is the Java bytecode
- next, we run the bytecode with the JVM (Java virtual machine)
  - `java HelloWorld` from the console window runs out program!
  - the JVM executes the `main` method of our program

Running hello world

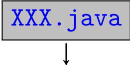
```
/* Java hello world */
public class HelloWorld{
  public static void main(String[] args){
    System.out.println("hello, world!");
  }
}
```

- Java convention is that
  - class name is capitalized (use camel case if more than one word)
  - class `XXX` must be in the file `XXX.java`
  - so `HellowWorld` must be in the file `HelloWorld.java`
- first we need to compile the source code into Java bytecode
  - IDE will have a compile button
  - `javac HelloWorld.java` from console window (shell)
  - this creates `HelloWorld.class`, which is the Java bytecode
- next, we run the bytecode with the JVM (Java virtual machine)
  - `java HelloWorld` from the console window runs out program!
  - the JVM executes the `main` method of our program

Running Java programs

`XXX.java`

Running Java programs



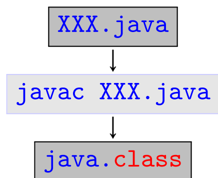
Running Java programs

javac XXX.java

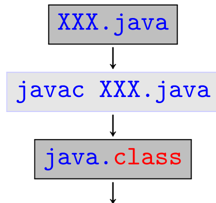
Running Java programs

```
XXX.java
   ↓
javac XXX.java
   ↓
```

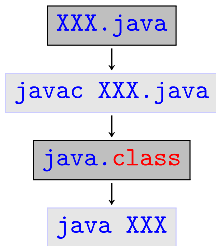
Running Java programs



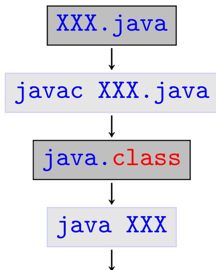
Running Java programs



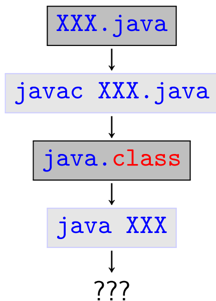
Running Java programs



Running Java programs



Running Java programs



Running Java programs

All Java **programs** must have a main method

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

- this is the entry point of every program
  - JVM calls the main method then you execute `java ClassName`
- any deviation and it is not a program
  (the name "`args`" can be changed)
- A Java program is also a Java `class`
  - we can run the program
  - we can create objects of the class

Variables

Variables are names given to locations in memory that are reserved to store data. They help us to keep track of the data in our code.

There are several kinds of variables in Java

instance variables ▸ store the state of an individual object

class variables ▸ store the state of a class
(these use the `static` modifier)

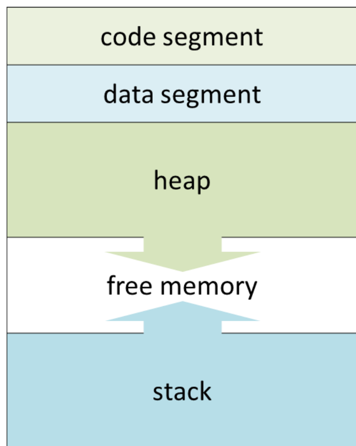
local variables ▸ store data that is used within a method. The variable does not exist when the method ends.

parameters ▸ store data that is passed into a method.

We will use a memory model to help us keep track of the data in our programs.

Memory Model

It will be very important to keep a mental model of where everything is stored in our programs.



Memory Model

It will be very important to keep a mental model of where everything is stored in our programs.

code segment ▸ program instructions

data segment ▸ data of class variables, literals

heap ▸ dynamically allocated memory

free space ▸ unused memory that is available for the heap or stack

stack ▸ function call stack

Memory Model

It will be very important to keep a mental model of where everything is stored in our programs.

We will usually use a **box and arrow** diagram to visualize the memory

more on this to come...

Primitive data types

in Java, everything is a **reference data type**

Primitive data types

in Java, everything is a **reference data type**

except for the 8 things that aren't!

Primitive data types

in Java, everything is a **reference data type**

except for the 8 things that aren't!

Java has eight **primitive data types**

- `byte`, `short`, `int`, `long` (integers)
- `float`, `double` (approximate real numbers)
- `boolean` (logical true/false)
- `char` (unicode characters)

Primitive data types

in Java, everything is a **reference data type**

except for the 8 things that aren't!

Java has eight **primitive data types**

- `byte`, `short`, `int`, `long` (integers)
- `float`, `double` (approximate real numbers)
- `boolean` (logical true/false)
- `char` (unicode characters)

What do all of these have in common?

Primitive data types

in Java, everything is a **reference data type**

except for the 8 things that aren't!

Java has eight **primitive data types**

- `byte`, `short`, `int`, `long` (integers)
- `float`, `double` (approximate real numbers)
- `boolean` (logical true/false)
- `char` (unicode characters)

What do all of these have in common?

- they are all just values (they have state)
- variables actually store the data

Primitive data types

there are four primitive data types for exact integers

- byte ▸ 8-bit signed integers

$$-128 \rightarrow 127$$

- short ▸ 16-bit signed integers

$$-32,768 \rightarrow 32,767$$

- int ▸ 32-bit signed integers

$$-2,147,483,648 \rightarrow 2,147,483,647$$

- long ▸ 64-bit signed integers

$$-9,223,372,036,854,775,808 \rightarrow 9,223,372,036,854,775,807$$

Primitive data types

there are two primitive data types for approximate decimal numbers (approximation to real numbers)

float
- 32-bit IEEE 754 floating point
  - 1 bit for sign, 8 bits for exponent, 23 bits for fraction accuracy

double
- 64-bit IEEE 754 floating point
  - 1 bit for sign, 11 bits for exponent, 52 bits for fraction accuracy

Primitive data types

there are two primitive data types for approximate decimal numbers (approximation to real numbers)

float
- 32-bit IEEE 754 floating point
  - 1 bit for sign, 8 bits for exponent, 23 bits for fraction accuracy
  - about 7 decimal digits of accuracy

double
- 64-bit IEEE 754 floating point
  - 1 bit for sign, 11 bits for exponent, 52 bits for fraction accuracy
  - about 15-16 decimal digits of accuracy

Danger! Computing with floats/doubles needs special care.

Primitive data types

and two more....

- boolean: `true` or `false`
- char: 16-bit Unicode character

  `char`s enclosed in single quotes

  - `‘x’`, `‘3’`, `‘Q’`, etc
  - from `‘\u0000’` (zero) to `‘\uffff’` (65,635)
  - unlike other languages, char is not equivalent to a byte (8-bits)

Primitive data types

Variables that store primitive data types store the actual data in memory where the variable is located.

For example,

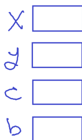
```
int y;                // variable declaration
int x = 12;           // declaration and initiazation
char c = 'a';
boolean b = true;
```

Primitive data types

Variables that store primitive data types store the actual data in memory where the variable is located.

For example,

```
int y;                  // variable declaration
int x = 12;             // declaration and initiazation
char c = 'a';
boolean b = true;
```



Primitive data types

Variables that store primitive data types store the actual data in memory where the variable is located.

For example,

```
int y;              // variable declaration
int x = 12;         // declaration and initiazation
char c = 'a';
boolean b = true;
```

x | 12
y | 0
c | 'a'
b | true

Objects

Variables that store objects are reference data types.
They store **references** of where the object data is in memory.

For objects, we declare variables, allocate memory for the object using `new`, and then define/initialize the object itself. All objects need this allocation step (sometimes implicit).

```
House h1;                          // declaration
h1 = new House();                  // memory allocation
h1.address = "123 Sesame Street";  // definition
```

```
House h2 = new House("123 Sesame Street");
```

The `new` operator allocates memory on the **heap** to store the object. It returns a reference to the location in heap where the memory is allocated. (Java hides this from you though.)

Objects

```
House h1;                                    // declaration
h1 = new House();                            // memory allocation
h1.address = "123 Sesame Street";            // definition
```

Objects

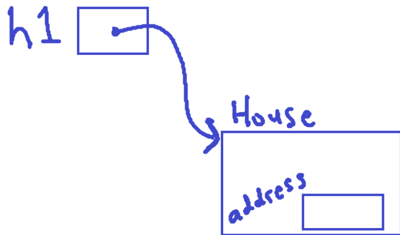
```
House h1;                          // declaration
h1 = new House();                  // memory allocation
h1.address = "123 Sesame Street";  // definition
```

h1

House h1;

Objects

```
House h1;                            // declaration
h1 = new House();                    // memory allocation
h1.address = "123 Sesame Street";   // definition
```

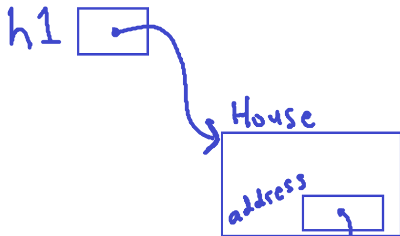


House h1;

h1 = new House() ;

Objects

```
House h1;                                // declaration
h1 = new House();                        // memory allocation
h1.address = "123 Sesame Street";        // definition
```



House h1;

h1 = new House();

h1.address = "---"

"123 Sesame street"

Strings

Strings are an essential data type that allow us to easily represent text. String are **immutable** in Java. Once you create a string you cannot change it. You create a new (different) string if you want to change it instead.

Because strings are used so much, Java allows for an easier allocation. Both of these create a variable that reference a string "cat".

```
String s1 = new String("cat");
String s2 = "cat";
```

Danger! The way strings are stored in memory is a bit tricky. We will come back to this soon.

Arrays

Arrays are an essential sequential container data type. They allow us to store a collection of items of the same type using a single name (variable) and index position (where a given item is in the collection).

```
Integer[] numbers;                        // variable declaration
numbers = new Integer[132];               // array allocation
for(int i=0; i<numbers.length; i+=1){
  numbers[i] = new Integer(i);            // fill the array
}
```

In this example, the `new` operator allocates enough memory to sequentially store 132 references to Integer objects.

Arrays

```
Integer[] n1;                              // array variable
long[] n2;                                 // declarations
n1 = new Integer[3];                       // array allocation
n2 = new long[3];                          // in the heap
for(int i=0; i<numbers.length; i+=1){
  n1[i] = new Integer(i);                  // fill the arrays
  n2[i] = i;                               with data
}
```

Arrays

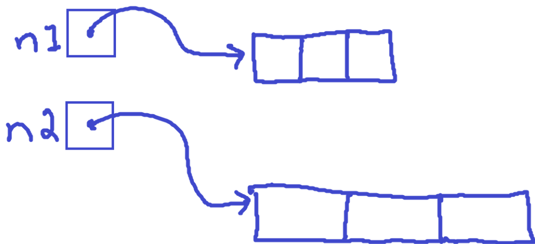
```
Integer[] n1;                                // array variable
long[] n2;                                   // declarations
n1 = new Integer[3];                         // array allocation
n2 = new long[3];                            // in the heap
for(int i=0; i<numbers.length; i+=1){
  n1[i] = new Integer(i);                    // fill the arrays
  n2[i] = i;                                 with data
}
```

n1

n2

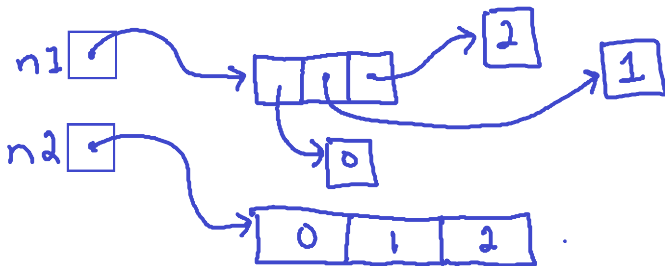
Arrays

```
Integer[] n1;                                  // array variable
long[] n2;                                     // declarations

n1 = new Integer[3];                           // array allocation
n2 = new long[3];                              // in the heap

for(int i=0; i<numbers.length; i+=1){
  n1[i] = new Integer(i);                      // fill the arrays
  n2[i] = i;                                   with data
}
```



Arrays

```
Integer[] n1;                                          // array variable
long[] n2;                                             // declarations
n1 = new Integer[3];                                   // array allocation
n2 = new long[3];                                      // in the heap
for(int i=0; i<numbers.length; i+=1){
  n1[i] = new Integer(i);                              // fill the arrays
  n2[i] = i;                                           with data
}
```



Objects

Every time you call the `new` operator you are allocating memory on the heap.

When you are done with that memory, Java will automatically de-allocate it (garbage collection). In other languages (like C and C++) you need to de-allocate memory yourself.

Operators

Operators are essentially functions with special syntax.

They are built-in to Java. Most operators are binary (have two inputs, called operands). Some are unary (one operand) and one is ternary (three operands).

- **assignment**
  - assigns the value on its right to the operand on its left (`=`)
  - compound assignment `+=`, `-=`, `*=`, `/=`, `%=`, ...
- **arithmetic**
  - basic arithmetic operations (`+`, `-`, `*`, `/`, `%`)
- **unary**
  - negation, logical negation, increment, decrement (`-`, `!`, `++`, `--`)
- **equality/relational**
  - equality/inequality (`==`, `!=`)
  - comparisons (`<=`, `>=`, `>`, `<`)
- **logic**
  - logical `and`, `or` (`&&`, `||`), ternary operator (`?:`)
- **bitwise**
  - `&`, `|`, `^`, `~`, `<<`, `<<<`, `>>`, `>>>`
- **other**
  - `new`, `instanceof`, `.`, `[]`, `()`

Operator precedence

order matters!

recall BEDMAS from elementary school

- brackets
- exponents
- division and multiplication
- addition and subtraction

Operator precedence

Operator	Description	Level	Associativity
[] . () ++ --	access array element access object member invoke a method post-increment post-decrement	1	left to right
++ -- + - ! ~	pre-increment pre-decrement unary plus unary minus logical NOT bitwise NOT	2	right to left
() new	cast object creation	3	right to left
* / %	multiplicative	4	left to right
+ - +	additive string concatenation	5	left to right
<< >> >>>	shift	6	left to right
< <= > >= instanceof	relational type comparison	7	left to right
== !=	equality	8	left to right
&	bitwise AND	9	left to right
^	bitwise XOR	10	left to right
|	bitwise OR	11	left to right
&&	conditional AND	12	left to right
||	conditional OR	13	left to right
?:	conditional	14	right to left
= += -= *= /= %= &= ^= |= <<= >>= >>>=	assignment	15	right to left

Type conversion

- automatic type conversion
  - JVM will do some conversions for you
- explicit type conversion
  - cast operator `(type) value`
  - use a method from a class

Type conversion

- automatic type conversion
  - JVM will do some conversions for you
- explicit type conversion
  - cast operator `(type) value`
  - use a method from a class

expression	expression value
`Integer.parseInt("32")`	
`(int) 4.99999`	
`Math.round(2.1)`	
`11 * 0.3`	
`(int) 11 * 0.3`	
`11 * (int) 0.3`	
`(int) (11*0.3)`	
`"cat" + 22`	

Type conversion

- automatic type conversion
  - JVM will do some conversions for you
- explicit type conversion
  - cast operator `(type) value`
  - use a method from a class

expression	expression value	
`Integer.parseInt(“32”)`	32	int
`(int) 4.99999`		
`Math.round(2.1)`		
`11 * 0.3`		
`(int) 11 * 0.3`		
`11 * (int) 0.3`		
`(int) (11*0.3)`		
`“cat” + 22`		

Type conversion

- automatic type conversion
  - JVM will do some conversions for you
- explicit type conversion
  - cast operator `(type) value`
  - use a method from a class

expression	expression value	
`Integer.parseInt(“32”)`	32	int
`(int) 4.99999`	4	int
`Math.round(2.1)`		
`11 * 0.3`		
`(int) 11 * 0.3`		
`11 * (int) 0.3`		
`(int) (11*0.3)`		
`“cat” + 22`		

Type conversion

- automatic type conversion
  - JVM will do some conversions for you
- explicit type conversion
  - cast operator `(type) value`
  - use a method from a class

expression	expression value	
`Integer.parseInt(“32”)`	32	int
`(int) 4.99999`	4	int
`Math.round(2.1)`	2	int
`11 * 0.3`		
`(int) 11 * 0.3`		
`11 * (int) 0.3`		
`(int) (11*0.3)`		
`“cat” + 22`		

Type conversion

- automatic type conversion
  - JVM will do some conversions for you
- explicit type conversion
  - cast operator `(type) value`
  - use a method from a class

expression	expression value	
`Integer.parseInt(“32”)`	32	int
`(int) 4.99999`	4	int
`Math.round(2.1)`	2	int
`11 * 0.3`	3.3	double
`(int) 11 * 0.3`		
`11 * (int) 0.3`		
`(int) (11*0.3)`		
`“cat” + 22`		

Type conversion

- automatic type conversion
  - JVM will do some conversions for you
- explicit type conversion
  - cast operator `(type) value`
  - use a method from a class

expression	expression value	
`Integer.parseInt("32")`	32	int
`(int) 4.99999`	4	int
`Math.round(2.1)`	2	int
`11 * 0.3`	3.3	double
`(int) 11 * 0.3`	3.3	double
`11 * (int) 0.3`		
`(int) (11*0.3)`		
`"cat" + 22`		

Type conversion

- automatic type conversion
  - JVM will do some conversions for you
- explicit type conversion
  - cast operator `(type) value`
  - use a method from a class

expression	expression value	
`Integer.parseInt(“32”)`	32	int
`(int) 4.99999`	4	int
`Math.round(2.1)`	2	int
`11 * 0.3`	3.3	double
`(int) 11 * 0.3`	3.3	double
`11 * (int) 0.3`	0	int
`(int) (11*0.3)`		
`“cat” + 22`		

Type conversion

- automatic type conversion
  - JVM will do some conversions for you
- explicit type conversion
  - cast operator `(type) value`
  - use a method from a class

expression	expression value	
`Integer.parseInt("32")`	32	int
`(int) 4.99999`	4	int
`Math.round(2.1)`	2	int
`11 * 0.3`	3.3	double
`(int) 11 * 0.3`	3.3	double
`11 * (int) 0.3`	0	int
`(int) (11*0.3)`	3	int
`"cat" + 22`		

Type conversion

- automatic type conversion
  - JVM will do some conversions for you
- explicit type conversion
  - cast operator `(type) value`
  - use a method from a class

expression	expression value	
`Integer.parseInt("32")`	32	int
`(int) 4.99999`	4	int
`Math.round(2.1)`	2	int
`11 * 0.3`	3.3	double
`(int) 11 * 0.3`	3.3	double
`11 * (int) 0.3`	0	int
`(int) (11*0.3)`	3	int
`"cat" + 22`	"cat22"	String

Expressions, statements, blocks

expressions
: A combination of variables, operators and function calls that evaluate to a single value

statements
: Forms a complete unit of execution

- **expression statements** include assignments, using `++` or `--`, function calls, object creation; each being terminated by a semicolon (`;`)
- **declaration statements** declare variables (specifies type and name and ends with semicolon)
- **control flow statements** alter the flow of execution of your program (more to come)

blocks
: ▸ a group of zero or more statements enclosed in curly braces `{ }`

Java control flow

Branching

- ```
  if(condition){
    <statements>
  }
  ```

- ```
  if(condition){
    <statements>
  }else{
    <statements>
  }
  ```

- ```
  if(condition1){
    <statements>
  }else if(condition2){
    <statements>
  }else{
    <statements>
  }
  ```
  ```

Java control flow

Branching

- one `if`
- followed by zero or more `else if`
- followed by zero or one `else`

Notes:

- you do not need curly braces `{}` for a single statement
  (it is safer to always put them!)

Java control flow

Branching

- ```
  int day = 3;
  ...
  switch(day){
    case 1: dayName = "Monday";
            s = s + 1;
            break;
    case 2: dayName = "Tuesday";
    case 3: dayName = "Wednesday";
            break;
    default:
            dayName = "None";
  }
  ...
  System.out.println(dayName);
  ```
  ```

Java control flow

Branching

In Java you can switch on

- `byte`, `short`, `char` and `int` primitive data types
- `Byte`, `Short`, `Character` and `Integer` classes
- `String` class and `Enum` types

Danger! Once a case has been satisfied all code after it is executed unless a `break` is found.

Java control flow

Repetition (Guarded)

- ```
  // guarded loop
  while(condition){
    <statements>
  }
  ```

- ```
  // guarded loop
  do{
    <statements>
  } while(condition);
  ```

Java control flow

Repetition (Bounded)

- ```
  // bounded loop
  for(initialization; termination; increment){
    <statements>
  }
  ```

- ```
  // enhanced for loop
  int[] characters = {'a', 'b', 'c'};
  for(int c : characters){
    <statements>
  }
  ```

Java control flow

Other branching statements

break Exits the innermost `for`, `while`, `do-while` or `switch` statement.

A labelled break can exit an outer statement.

continue Skips the current iteration of the innermost `for`, `while` or `do-while` loop.

A labelled continue can skip the iteration of an outer loop.

return Ends the current method and brings control back to the place where the method was called.
(possibly passes back a value)

Procedural programming

- program is a sequence of function calls
- helps us write structured programs
  - this helps `us` write better programs (Software Engineering)
- code that does one thing is grouped into a procedure (function/method)
  - helps us achieve modularization
- code that is repeated during the execution of the program is put in a procedure (function/method)
  - lets us write less code (less testing and maintenance!))

You have already been doing this in COMP1005/1405.

Procedural programming

```
public class Game{
  static boolean play(int lives){
    int speed = 3;
    ...
  }

  public static void main(String[] args){
    int lives = 3;
    boolean keepPlaying = true;
    while(lives > 0 && keepPlaying){
      play(lives);
    }
  }
}
```

Procedural programming

Basic I/O (input/output)

- ```
  System.out.println("hi there!");
  System.out.print("hi ");
  System.out.print("hi \n");
  ```
- ```
  Scanner keyboard = new Scanner(System.in);
  String input = keyboard.next();
  System.out.println(input);
  ```

Command Line Arguments

```
/* Java hello world */
public class HelloWorld{
  public static void main(String[] args){
    System.out.println("hello, world!");
  }
}
```

- the JVM calls your program's `main` method
- `main` has input parameters
- we can pass arguments into `main` in two ways
  - specify them when you run your program (command line arguments)
  - call `main` from within another program

Command Line Arguments

```
/* Java hello world */
public class HelloWorld{
  public static void main(String[] args){
    System.out.println("hello, world!");
  }
}
```

- the JVM calls your program's `main` method
- `main` has input parameters
- we can pass arguments into `main` in two ways
  - specify them when you run your program (command line arguments)
  - call `main` from within another program

Command Line Arguments

```
/* Java hello world */
public class HelloWorld{
  public static void main(String[] args){
    System.out.println("hello, world!");
  }
}
```

- the JVM calls your program's `main` method
- `main` has input parameters
- we can pass arguments into `main` in two ways
  - specify them when you run your program (command line arguments)
  - call `main` from within another program

Command Line Arguments

```
/* Java hello world */
public class HelloWorld{
  public static void main(String[] args){
    System.out.println("hello, world!");
  }
}
```

- the JVM calls your program's `main` method
- `main` has input parameters
- we can pass arguments into `main` in two ways
  - specify them when you run your program (command line arguments)
  - call `main` from within another program

Command Line Arguments

```
/* Java hello world */
public class HelloWorld{
  public static void main(String[] args){
    System.out.println( args[0] );
  }
}
```

Running the program from a shell as

```
java HelloWorld Hi There!
```

will output

```
Hi
```

Command Line Arguments

```
/* Java hello world */
public class HelloWorld{
  public static void main(String[] args){
    System.out.println( args[0] );
  }
}
```

Running the program from a shell as

```
java HelloWorld Hi There!
```

will output

```
Hi
```

Assignment 1

- A good bit of reading
- Remember that this is a condensed course
  (each assignment is like 2 assignments)

  start early!
- Check the grading rubric