What Is An Object?

An entity that exists in an operating computer program that has
State, behaviour, identity

- The STATE of an Object is the collection of **information held in that object**. This information **may change** over time, as a result of operations carried out on the Object
- The BEHAVIOUR is the **collection of operations** that an Object **supports**
- The IDENTITY of an Object allows the program **access** to a specific Object

- An Object represents **real** or **abstract entities**

- An Object representing a real entity for example: State: awake, hungry, purring. Behaviour: scratchingSofa Identity: ginger

What is a Class?

- You can have many (possibly infinite) different Objects with different values for each State category (or ATTRIBUTE)
- But if each of these Objects has the same set of possible behaviours then you can group these Objects together into a Class

- A class is defined in the source code of a program: It defines…
  - The operations that are allowed on instances of this class (the methods)
  - The possible categories of state that are allowed for instances of this class (the attributes)
- Instances of a class are created when the program is running

- The State of an Object instance can be partly or completely defined when the instance is created
- The State will likely change when operations are carried out on the instance
- However, **attributes cannot be added or removed** and **behaviour cannot be added or removed**. These are **defined in the Class**

Object Categories

In a program, Objects will probably fall into one of these general categories:

- Tangible things (ex. Cat)
- Agents (StringTokenizer)
- Events and transactions (MouseEvent)
- Users and roles (Administrator)
- Systems (MailSystem)
- System interfaces and devices (File)
- Foundational classes (String)

Object Extremes

Two extremes of object structure:

Utility Classes: All STATIC methods and attributes

You do not instantiate these classes, there is not point
The Math class for example

Customizable Classes: All NON-STATIC methods and attributes

Attribute values (some or all) must be set at the time of instantiation before the class can be used
Scanner Class for example

- Many classes fall **in-between** these two extremes:
  - A mix of static and non-static methods
  - static methods have nothing to do with the attributes and so can be used without instantiation of the class
  - Non-static methods depend on the attributes which must be set through instantiation
  - Wrapper classes for example: Double, Integer, etc.

Encapsulation

Is the process of defining a Class that has at least one customizable attribute

- Encapsulation is about the **abstraction** or **containment** of the attributes defined in the class
- In Java, methods and attributes **must** be encapsulated or contained in a class definition

The Best Object Design:

- Supports the re-usability of code
- Provides a well-defined interface to other objects and the user(s)
- Builds into an Object the code that ensures the **integrity** of the data stored in the Object by:
  - Making sure that initial data values are “legal”
  - Controlling (or Preventing) changes to data
  - Preventing “Privacy Leaks” when data is returned out of an Object
- Works well with modular design practices making code easier to write, test, and debug

- This is the driving principle for the design for all Objects in the Java API!
- **Attributes must be declared private, so that the class that owns them can control how they are set**

If attributes are private then how can a user of the class assign the values of the attributes?

Through methods of course!
Constructor(s) (when instantiated only)
Mutator(s) (anytime after instantiation)

- Within these methods, you can write code to check the parameters for validity
- If values are NOT LEGAL, throw an exception!

Defining Exceptions

- You can throw an exception already defined in Java but, Most of the time you will want to throw your own exception objects
- Keywords: extends, super, throw, throws

Assigning Private Attributes - Constructors

- Constructors are special methods that have the same name as the class in which they reside, but have NO RETURN TYPE (not even void)
- They MUST be public
- They are only executed ONCE, when the object is being instantiated. You can’t invoke them later
- Constructors are often OVERLOADED, which means you can have MANY constructors with different parameter lists

- Note that once you have written a constructor with parameters, you can no longer use a constructor without any parameters, called the default constructor
- If you want an empty constructor, you must write one yourself, in addition to the parameterized constructor(s)

Suppose you don't want to force the user to supply all parameters when he instantiates the object. How are you going to do this? Overload the constructors

Suppose you want to edit parameters after you have created the object. How are you going to do this? Provide mutator(s)

Preventing Instantiation
- Provide ONLY the DEFAULT CONSTRUCTOR and make it PRIVATE
- Used by math class for example, there is no need to instantiate the Math class since all its methods and attributes are STATIC

Assigning Private Attributes - Mutator Methods
- Called "set" methods — in fact, by convention the word "set" is the first word in the method name
- These methods must also check for legal parameters
- Usually the constructor invokes mutators, when they exist

The "this" Thing
- Used by one constructor to invoke the other one
- Can be used to supply a reference to the current object in that object's code
- this means myself
- Another use, suppose I used the same attribute name as a constructor parameter — year
  In constructor I would need to say: this.year = year;

Accessors
- Accessor methods return the value of an attribute
- By convention, accessor methods start with the word "get"
- One accessor per attribute

Ex.

```
public int getNumMunchkins() {
        return numMunchkins;
}//end getNumMunchkins Accessor
```

- They are pretty simple methods except when you are returning an object

Useful Java Classes

Classes defined in java.lang package are automatically imported as they are used quite often
- The Wrapper classes
- Math
- Object
- String
- System
- Thread

WMOSST

static Methods
- static attributes and methods are loaded once into memory and not garbaged collected until main is FINISHED
- Generally, they are utility methods that do not depend on the values of a class' attributes
- static methods can be invoked WITHOUT instantiation of the Object that owns them
- static methods and attributes are SHARED by ALL instances of a class - There is only ONE copy of these methods in memory
- static method can only invoke other static methods in its own class, can't have pieces of code disappearing from a static method in memory

Math Class
- A collection of **static constants** and **static mathematical methods**
- you cannot instantiate the Math class

Wrapper Classes
- Sometimes its necessary for a PRIMITIVE TYPE value to be an OBJECT, rather than just a primitive type
  - Some data structures only store Objects
  - Some Java methods only work on Objects
- Each primitive type has an associated Wrapper class:

char	Character
int	Integer
long	Long
float	Float
double	Double

- Each wrapper class Object can hold the value that would normally be contained in the primitive type variable, but now has a number of useful static methods

Ex. Integer number = new Integer(46); //wrapping
int aNumber = number.intValue(); //aNumber is 46

- The Character wrapper class: has methods to convert between ASCII and Unicode numeric values and characters
  - isDigit(character) returns true if character is a digit
  - isLetter(character)
  - isUpperCase(character)
  - isLowerCase(character)
  - isWhitespace(character)
  - toLowerCase()
  - toUpperCase()

System Class
- Have used System.out.println(), print(), printf(), and err.println()

Other Useful System Class Methods:

System.currentTimeMillis()
    Returns, as a long, the number of milliseconds elapsed since midnight 01/01/1970

System.exit(0)
    Immediate termination of program

System.getProperties()
    All kinds of system specific info
System.getProperty(string)
    Displays single system property
System.nanoTime()
    Time in nanoseconds

String Class
- Since Strings are Objects they can have methods
- There are 67 String methods
    - length()
    - equals(OtherString)
    - equalsIgnoreCase(OtherString)
    - toLowerCase()
    - toUpperCase()
    - trim()
    - charAt(Position)
    - substring(Start)
    - substring(Start, End)
    - IndexOf(SearchString)
    - replace(oldChar, newChar)
    - startsWith(PrefixString)
    - endsWith(SuffixString)
    - valueOf(integer)
- String's DO NOT HAVE ATTRIBUTES

- Constructors are special methods with the same name as the class that are invoked when you instantiate the class
- The String class has 15 of these — the constructor is "overloaded"

- Strings are **immutable** meaning that they cannot be altered, only re-assigned
- There are no methods that can alter characters inside a string while leaving the rest alone
- Arrays are **mutable** in contrast, any element can be changed

Mutability
- For an array you can use [ ] on either side of an assignment operator
- A String cannot use [ ] to access individual characters, only the .charAt() method
- The .charAt() method cannot be used on left hand side of assignment operator

Other java.lang Classes
- Object is the base class for all objects in java
- Thread is a base class used to create threads in multi-threaded program.
- A class not in java.lang:
    String Tokenizer Class
    - This useful class is in the "java.util" package, so you need to import java.util.*
    - This class provides an easy way of parsing strings up into pieces, called "tokens"
    - Tokens are separated by "delimiters" that you can specify or you can accept a list of default delimiters
    - The constructor method for this class is overloaded

- So, when you create an Object of type StringTokenizer, you have 3 options
    new StringTokenizer(String s)
    new StringTokenizer(String s, String delim)
    new StringTokenizer(String s, String delim, boolean returnTokens)
- s is the String you want to "tokenize"
- delim is a list of delimiters, by default it is: \t\n\r
- You can specify your own list of delimiters if you provide a different String for the second parameter
- If you specify a **true** for the final parameter, then delimiters will also be provided as tokens
- Default is false, delimiters are not provided as tokens
- Note that the StringTokenizer object is emptied out as tokens are removed from it
- You will need to re-create the object in order to tokenize it again

Method Overloading

- A method can have the same name in many different classes
- "Overloading" is when a **method name is used more than once** in method declarations with the **same class**
- The rule is that no two methods with the same name within a class can have the same number and/or types of parameters in the method declarations (the "NOT" rule)
- Its for convenience!
- Java does not have default arguments
- Allows the user to call a method without requiring him to supply values for all the parameters
- One method name can be used with many different types and combinations of parameters
- Allows the programmer to keep an old method definition in a class for "backwards compatibility"

How does it work?

- Java looks through all methods until the parameter types match with the list of arguments supplied by the user. If none match, Java tries to cast types in order to get a match. (Only "widening" casting like int to double)

- You can have as many overloaded method definitions as you want, as long as they are differentiated by the type and/or number of the parameters listed in the definition
- **Do not change the return type, its tacky!!**

Exceptions

How can a method indicate that it is unable to return what it is supposed to return?
How can a method deliver details about the error condition?
How else can you prevent the instantiation of an Object?

- The limitation of only returning a single "thing" means that you either designate error values for the "thing" or you have some other way to return the indication of an error
- The designers of Java followed conventions used by many other OOP languages — they allowed for another way to get something out of a method. However, an **exception is thrown, not returned**

- **Exceptions are Objects**

- When an error condition is encountered, a method can throw an instance of a pre-defined exception Object
- A method can throw several exceptions, one for each possible kind of error condition

Exceptions: Propagation
- If a method throws an exception, then that method is immediately halted and there is no need for any return value, even if the method is non-void

- The invoking method then receives the exception, if it does not catch it, then it goes to the next invoking method — all the way to main, if necessary (Called cascading or propagation)
- Finally, if main does not catch the exception, your program crashes and a message is sent tot the console window

Exceptions: Message Handler
- How does an Exception Object contain information about the error condition?
  - The type of the Object:
    - IOException
    - NumberFormatException
    - FileNotFoundException
    - ArrayIndexOutOfBoundsException etc…
- So the method should throw a relevant exception object
- Turns out that exceptions can also carry a String message

- An exception is just a “Bearer of Bad Tidings”
- The exception itself cannot do anything about the problem, but it can act to stop your program

Exceptions: Catching
- This is done with a “try/catch” block
- The compiler will force you to use try/catch block when you invoke methods that throw exceptions

Syntax:

```
try{
        //block of statements that might generate an exception
} catch (exception_type identifier) {
        //block of statements
} [ catch (exception_type identifier) {
        //block of statements …
} ] [ finally {
        // block of statements
}]
```

- You **must have at least one catch block** after the “try block” (otherwise the try block would be useless!)
- You can have many catch blocks, one for each exception you are trying to catch
- The code in the “finally” block is ALWAYS executed, whether an exception is thrown, caught, or not

Checked vs. Unchecked Exceptions

- Checked exceptions must either be **caught in a method** or **thrown from that method** (using the throws clause in the method header)
- You should not catch an un-checked exception or an error
- The compiler will ensure that checked exceptions are handled properly
- Unchecked exceptions occur only at runtime and the compiler does not care about them
- Unchecked exceptions are all sub-classes of java.lang.Error

Try with Resources

- This new syntax is very useful with Java 7&8's improved file I/O syntax

Syntax:

```
try (instantiation; instantiation; …) {
        //other statements that might generate an exception
} [ catch (exception_type identifier) {
        //block of statements
} ] [ catch (exception_type identifier) {
        //block of statements …
} ] [ finally {
        //block of statements
} ]
```

- The instantiation(s) inside the set of ( ) immediately after the try keyword are declared resources that must be local to the try/catch block
- Note that there is no ; at the end of the list and that there does not have to be any catch blocks
- These resources must all implement the Autocloseable interface, which means that the try block can close the resource when it is finished
- Resources will be closed whether or not an exception is thrown because their scope is forced to be in the try block only
- As a result, all use of the resource must also take place in the try block

Scanner Class Tokenizer

- Scanner class has tokenizer built into it
- Scanner uses a regular expression (regex) instead of the (easier to understand, but less powerful) delimiter list
- default of regex is "\p{javaWhitespace} +" which means any number of whitespace characters
- A **whitespace character** is a space, tab, linefeed, formfeed, or carriage return

Aliasing Objects: Array Example

```
int[ ] first = {1,2,3,4,5}'
int[ ] second = {10,20,30,40,50,60,70};
```

(the two arrays will point to different addresses)

```
second = first; //aliasing
```

(the pointer to the first array address is then pointed to the second, where after garbage collection, all the information in second is REPLACED with first)

```
first[4] = 500; //second[4] is also 500
```

- So setting one array to equal another as in: array1 = array2; sets array1 to point to same data memory location that was (and still is) pointed to by array2
- Now changing the value of an element in array2, will change that same element in array1, or vice versa. This makes sense since both array Objects point to the same set of data values in memory!

Garbage Collection
- Some computer programming languages require you to indicate when you are done with variables so the memory they are occupying can be released back to the OS
- Fortunately, Java has an AUTOMATIC garbage collection system:

- variables are garbage collected one you **move outside their scope**
- Object contents are garbage collected when there are **no pointers pointing to the contents**

Aliasing Objects
- Passing an Object into a method results in the method's parameters being aliased to the Object passed (called Passing by Reference!)

Passing Parameters by Reference
Ex. in main:

```
int[ ] arrayA = {1,2,3,4,5};
passArray(arrayA); //invoke passArray
```

The passArray method:

```
public static void passArray(int[ ] arrayB) {
        // arrayB is aliased to arrayB from main
        // making elemental changes to arrayB will also change elements in arrayA in main
        arrayB[3] = 400;
}//end passArray
//arrayA[3] is 400 in main
```

The Rule for parameter passing into methods is:
Objects are passed by reference, primitive types are passed by value

- Only element by element changes in arrays will "stick"
- Re-assigning the array to a pointer that has local scope in a method will not "stick"
- If you make element by element changes using an aliased local pointer (like the parameter), changes **will** "stick"
- Strings are **immutable** so this does not apply. You cannot make elemental changes in a String, even though a String is passed by reference
- So, **mutable** Objects (like arrays) can be passed **into and out** of a method through the parameter list. If a method changes the contents of a mutable Object passed into it, those changes "stick" even when the method is complete

Comparing Objects
Testing arrays and Objects for equality (with the "==" boolean operator is interesting:
- This test will only give true when **both objects** have been **aliased**, using the assignment operator "="

- So even if both arrays have identical contents, "==" will return false, unless both arrays point to the same location
- This means that **comparing Objects** with **"==" only compares pointers not contents**

Standard Encapsulated Structure
- private attributes
- Constructor(s) could be overloaded
- Mutators
- Avoiding privacy leaks
- Constructors and mutators throw exceptions to prevent the creation of an illegal object
- Accessors
- Other standard members: toString, compareTo, equals, clone

Standard Methods — toString(), equals(), compareTo(), clone()

toString() Method
- If we try to print an Object, we will get a string composed of the object type and its has code (Halloween2@923e30)
- So to get more useful view of the contents of the Object, defined a toString() method that returns a String
- This method will be called automatically whenever a String version of the object is need
- It overrides the toString() method from the parent Object class

equals() Method
- Accepts another object of type "Halloween4" and returns true if they are equal, false otherwise
- Usually all the attributes have to match
- Two ways to write an equals() method but both will return true or false
  - Accepts a Halloween4 object as a parameter
  - **Accept an Object as a parameter (Object is the base class for all objects in Java!)**
- The **PROPER way is the second one** — then you will overwrite the equals() method inherited from the base Object class
- The instance keyword checks if the supplied Object is indeed a Halloween4 object before it attempts to cast it (otherwise you get a ClassCastException)
- Then the object is cast to be of type Halloween4 using the casting operator
- Note how the method can refer directly to the attributes of the other object

compareTo() Method
- Compares a supplied object with the current one, based on your comparison criteria
- It returns an int value
- (like the compareTo() method in the String class)
- You have to decide on the basis for comparison
- The base Object class does not have this method, so you don't have to worry about writing one to take an Object as a parameter

clone() Method
- Suppose you want to make a completely separate copy of an int array called stuff:

```
int[ ] clonedStuff = stuff.clone();
```

OR

```
int[ ] clonedStuff = new int[stuff.length];
for (int i = 0; i < stuff.length; i++)
        clonedStuff[i] = stuff[i];
```

- Returns a deep copy of the current object
- A “deep copy” is not aliased in any way to the original object, but contains the same values for each attribute

Annotations

- The “@Override” thingy
- You don’t have to use these, but the pre-compiler likes them
- This one tells the pre-compiler that “we know we are overriding toString in the parent class and we are ok with it!