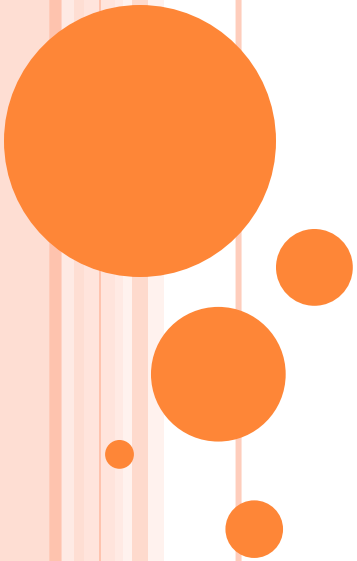
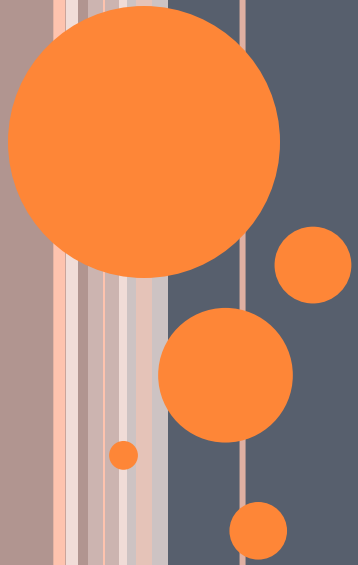


Java

Agenda

- Variable
- Encapsulation
- Class & Object
- Access Specifiers



Variables

What is a variable?

- named storage that our programs can manipulate
  - int a, b, c; // Declares three ints, a, b, and c.
  - int a = 10, b = 10; // Example of initialization
  - byte B = 22; // initializes a byte type variable B.
  - double pi = 3.14159; // declares and assigns a value of PI.
  - char a = 'a'; // the char variable a is initialized with value 'a'

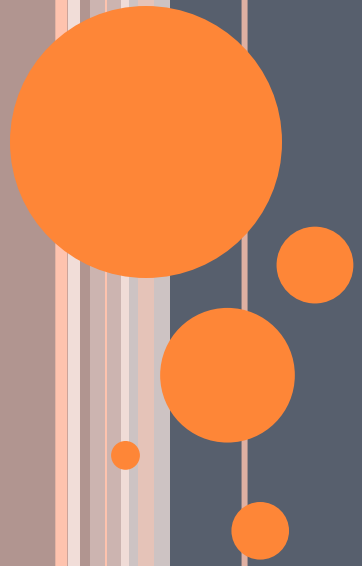
Variable types

- Local variables
- Instance variables
- Class/static variables

Local Variables

- Are declared in methods, constructors, or blocks.
- Are created when the method, constructor or block is entered and the variable will be destroyed once it exits the method, constructor or block.
- Access modifiers cannot be used for local variables.
- Local variables are visible only within the declared method, constructor or block.
- Local variables are implemented at stack level internally.
- There is no default value for local variables so local variables should be declared and an initial value should be assigned before the first use.

```
public class Test{
  public void Age(){
    int age = 0;
    age = age + 7;
    System.out.println("Age is : " + age);
  }

  public static void main(String args[]){
    Test test = new Test();
    test.Age();
  }
}
```



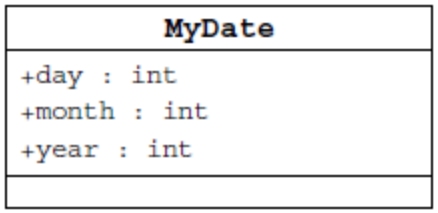
Encapsulation

What does Encapsulation mean?

- Refers to an object's ability to hide data and behavior that are not necessary to its user
- Enables a group of properties, methods and other members to be considered a single unit
- Encapsulation is also known as information hiding

Information Hiding

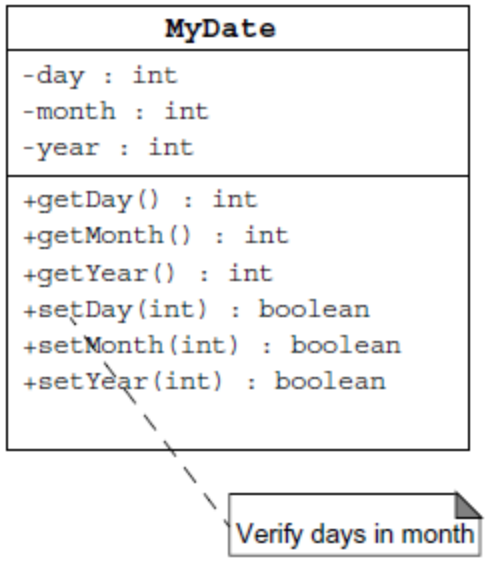
The problem:



Client code has direct access to internal data (d refers to a `MyDate` object):

```
d.day = 32;
// invalid day

d.month = 2; d.day = 30;
// plausible but wrong

d.day = d.day + 1;
// no check for wrap around
```

The solution:

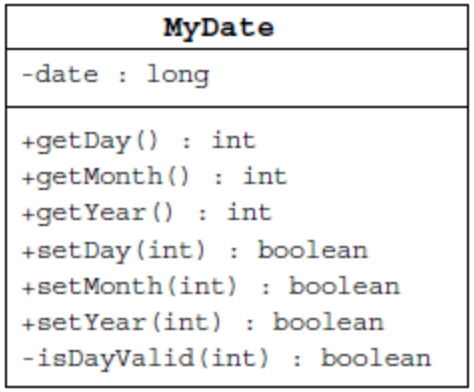


Client code must use setters and getters to access internal data:

```
MyDate d = new MyDate();

d.setDay(32);
// invalid day, returns false

d.setMonth(2);
d.setDay(30);
// plausible but wrong,
// setDay returns false

d.setDay(d.getDay() + 1);
// this will return false if wrap around
// needs to occur
```

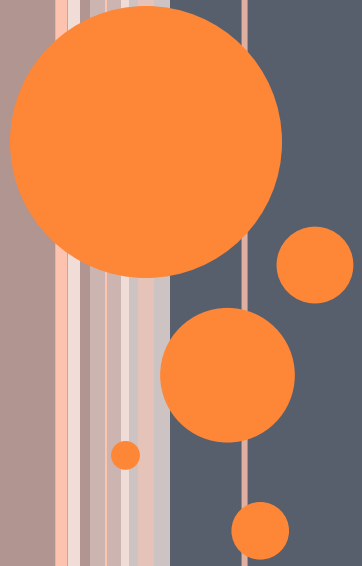
Encapsulation

- Hides the implementation details of a class
- Forces the user to use an interface to access data
- Makes the code more maintainable



BENEFITS OF ENCAPSULATION

- Protection of data from accidental corruption
- Specification of the accessibility of each of the members of a class to the code outside the class
- Flexibility and extensibility of the code and reduction in complexity
- Lower coupling between objects and hence improvement in code maintainability



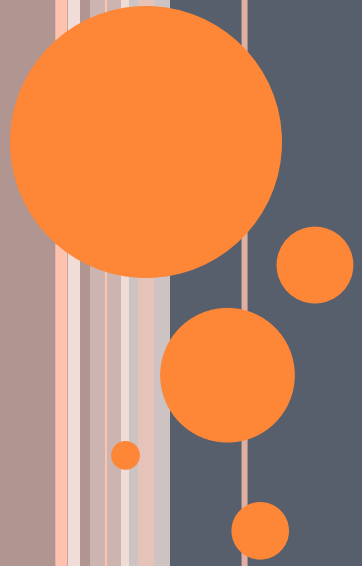
Class & Object

Class & Object

- A class is like a blueprint
- It defines the data and behaviour of a type
- Client code can use it by creating *objects* or i*nstances* which are assigned to a variable.
- The variable remains in memory until all references to it go out of scope.
- At that time, the JVM marks it as eligible for garbage collection.

Classes as Blueprints for Objects

- In manufacturing, a blueprint describes a device from which many physical devices are constructed.
- In software, a class is a description of an object:
  - A class describes the data that each object includes.
  - A class describes the behaviors that each object exhibits.
- In Java technology, classes support three key features of object-oriented programming (OOP):
  - Encapsulation
  - Inheritance
  - Polymorphism



Access Specifiers

Access Specifiers

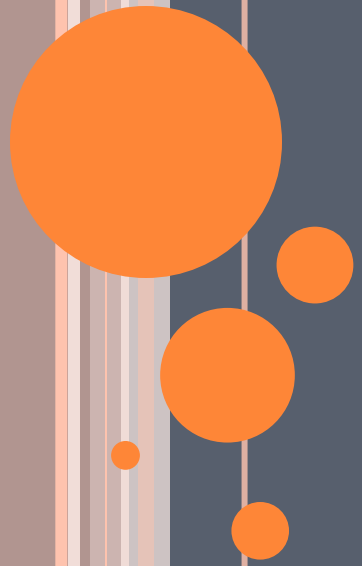
Modifier	Same Class	Same Package	Subclass	Universe
`private`	Yes			
default	Yes	Yes		
`protected`	Yes	Yes	Yes	
`public`	Yes	Yes	Yes	Yes

Access Control Modifiers

- Visible to the package, the default. No modifiers are needed.
- Visible to the class only (private).
- Visible to the world (public).
- Visible to the package and all subclasses (protected).

Non Access Modifiers

- The *static* modifier for creating class methods and variables
- The *final* modifier for finalizing the implementations of classes, methods, and variables.
- The *abstract* modifier for creating abstract classes and methods.
- The *synchronized* and *volatile* modifiers, which are used for threads.



Instance & static Variable

Instance Variable

- Declared in a class, but outside a method, constructor or any block.
- When space is allocated for an object in the heap, a slot for each instance variable value is created.
- Created when an object is created with the use of the keyword 'new' and destroyed when the object is destroyed.
- Hold values that must be referenced by more than one method or essential parts of an object's state that must be present throughout the class.
- Access modifiers can be given for instance variables.
- Instance variables have default values.
  - For numbers the default value is 0
  - for Booleans it is false
  - for object references it is null.
- Can be accessed directly by calling the variable name inside the class.

Class / static variables

- Class variables also known as static variables
- Declared with the *static* keyword in a class, but outside a method.
- There would only be one copy of each class variable per class, regardless of how many objects are created from it.
- Static variables are created when the program starts and destroyed when the program stops.
- Most static variables are declared public since they must be available for users of the class.
- Values can be assigned during the declaration or within the constructor. Additionally values can be assigned in special static initializer blocks.
- Static variables can be accessed by calling with the class name . *ClassName.VariableName*.
- When declaring class variables as public static final, then variables names (constants) are all in upper case. If the static variables are not public and final the naming syntax is the same as instance and local variables.