**Primitive Types**

**Primitive types**

- Primitive types: 
  - `byte`, `short`, `int`, `long`, 
  - `float`, `double`, `char`, `boolean`
- Example: 
  - `int size = 42;`
  - `size` is a primitive variable, i.e., a variable that contains a data value of the declared type.

**Limitations**

- All primitive types have limited ranges
  - Exceeding the range will cause inaccurate results
- `float` and `double` are typically approximations
  - don’t use `==` to compare real types
  - Use `Math.abs(x-y) <= EPSILON` where `EPSILON` is some small number

**Operations on primitive types**

- Variable holds value
- Assignment statements assign values
  - E.G., `int n = 10;`
- Operations are defined as language primitives
  - E.G., `n += 5;`
- Relational operators, etc., are defined as language primitives
  - E.G., `if (n <= 15) . . .`
Result of logical operators

| x     | y     | x && y | x || y | !x |
|-------|-------|--------|--------|----|
| false | false | false  | false  | true|
| false | true  | false  | true   | true|
| true  | false | false  | true   | false|
| true  | true  | true   | true   | false|

DeMorgan’s Laws

!(x && y) == !x || !y
!(x || y) == !x && !y

Short-circuit evaluation

```java
int x = 3;
int y = 5;
if ((x == y) && (y > 0)) ...
if ((x < y) || (y < 0)) ...
```

Reference types

- All types that are not primitive are reference or class types
- Example:
  ```java
  String greeting = "Howdy";
  >>> greeting is a reference variable, i.e., a variable that contain a reference to (address of) the memory location of the data.
  ```

Objects

- An object is a program entity that
  - contains data
  - performs certain actions
- The actions are called methods
- The actions of various objects interact to form a solution for a given problem
Classes

• A class defines the characteristics for all objects of its type
• A class gives a general description of
  – what an object of the type is (instance data)
  – what an object of the type can do (methods)

Objects / Classes

• An object is an instance of a class.
• A program may have many instances (objects) of one class.
• All instances of the same class have
  – The same kinds of data
  – The same methods

An Example of a Class

Three instances of the Automobile class

Instantiation

• Invoke a constructor
• Example:
  Automobile bobsCar = new Automobile("Sedan", 2000, 0.9, 55, 21405);

• Keyword new invokes a constructor for the Automobile class.
• Note: this example uses an “overloaded” constructor. The “default” constructor does not accept any parameters.

Object variables

• An object variable is used to store the address of (reference to) an object
• When a class is instantiated, the system creates an object variable, and stores the address of the new object therein.
• E.G., bobsCar (previous slide) is an object variable.
**Instantiation**

A memory cell that contains the memory address of the data for `bobsCar`

```
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>“Sedan”</td>
<td>2000</td>
<td>0.9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>55</td>
<td>21405</td>
</tr>
</tbody>
</table>
```

**Aliases**

- It is possible for two variables to reference the same object. They are aliases.
- Example:
  ```java
  Automobile myCar = bobsCar;
  ```

```
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>“Sedan”</td>
<td>2000</td>
<td>0.9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>55</td>
<td>21405</td>
</tr>
</tbody>
</table>
```

If an object has two (or more) aliases, the object may be referenced and/or modified through any of those aliases.

**Invoking methods in a program**

- Instead of calling a function to “do something to” an object (imperative programming), tell the object to perform one of its actions (object-based programming).
- Examples:
  ```java
  bobsCar.accelerate(5);
  int currentSpeed = bobsCar.getSpeed();
  if (bobsCar.getFuelLevel() < 0.1)
      bobsCar.decelerate(10);
  ```

**Arguments and Parameters**

- Invocation of a method must have same number of arguments as formal parameters in the declaration of the method.
- Arguments are associated in order with formal parameters.
- Argument types must be assignment-compatible with the associated formal parameter types.

**Valued methods** return a single value, and should be used in an expression.
- Return type of method must be assignment-compatible in the context of the call.

**void methods** do not return a value, and should be used as a single statement.

- Examples:
  ```java
  bobsCar.accelerate(5);
  int currentSpeed = bobsCar.getSpeed();
  if (bobsCar.getFuelLevel() < 0.1)
      bobsCar.decelerate(10);
  ```

**Arguments and Parameters**

- The formal parameter in the method becomes an alias for the argument passed to it, and is treated as a local variable.
- The formal parameter is discarded when the method terminates.
- Example: Suppose that a `matchSpeed` method is defined as follows:
  ```java
  public void matchSpeed(Automobile other) {
      int diff = getSpeed() - other.getSpeed();
      if(diff > 0) other.accelerate(diff);
      else other.decelerate(-diff);
  }
  ```
Before calling `matchSpeed`

```
Before calling matchSpeed

<table>
<thead>
<tr>
<th>automobile</th>
<th>model</th>
<th>year</th>
<th>fuel level</th>
<th>speed</th>
<th>mileage</th>
</tr>
</thead>
<tbody>
<tr>
<td>bobsCar</td>
<td>&quot;Sedan&quot;</td>
<td>2000</td>
<td>0.9</td>
<td>55</td>
<td>21405</td>
</tr>
<tr>
<td>suesCar</td>
<td>&quot;SUV&quot;</td>
<td>2001</td>
<td>0.45</td>
<td>35</td>
<td>9864</td>
</tr>
</tbody>
</table>
```

Call `matchSpeed`

```
bobsCar.matchSpeed(suesCar);
```

`suesCar` is associated with `other`

`other` is an alias of `suesCar`.

```
Call matchSpeed

<table>
<thead>
<tr>
<th>automobile</th>
<th>model</th>
<th>year</th>
<th>fuel level</th>
<th>speed</th>
<th>mileage</th>
</tr>
</thead>
<tbody>
<tr>
<td>bobsCar</td>
<td>&quot;Sedan&quot;</td>
<td>2000</td>
<td>0.9</td>
<td>55</td>
<td>21405</td>
</tr>
<tr>
<td>suesCar</td>
<td>&quot;SUV&quot;</td>
<td>2001</td>
<td>0.45</td>
<td>35</td>
<td>9864</td>
</tr>
<tr>
<td>other</td>
<td>&quot;Sedan&quot;</td>
<td>2000</td>
<td>0.9</td>
<td>55</td>
<td>21405</td>
</tr>
</tbody>
</table>
```

After executing

```
other.accelerate(diff);
```

`suesCar.speed` is changed

```
After executing

other.accelerate(diff);

<table>
<thead>
<tr>
<th>automobile</th>
<th>model</th>
<th>year</th>
<th>fuel level</th>
<th>speed</th>
<th>mileage</th>
</tr>
</thead>
<tbody>
<tr>
<td>bobsCar</td>
<td>&quot;Sedan&quot;</td>
<td>2000</td>
<td>0.9</td>
<td>55</td>
<td>21405</td>
</tr>
<tr>
<td>suesCar</td>
<td>&quot;SUV&quot;</td>
<td>2001</td>
<td>0.45</td>
<td>35</td>
<td>9864</td>
</tr>
<tr>
<td>other</td>
<td>&quot;Sedan&quot;</td>
<td>2000</td>
<td>0.9</td>
<td>55</td>
<td>21405</td>
</tr>
</tbody>
</table>
```

After `matchSpeed` terminates

```
After matchSpeed terminates

<table>
<thead>
<tr>
<th>automobile</th>
<th>model</th>
<th>year</th>
<th>fuel level</th>
<th>speed</th>
<th>mileage</th>
</tr>
</thead>
<tbody>
<tr>
<td>bobsCar</td>
<td>&quot;Sedan&quot;</td>
<td>2000</td>
<td>0.9</td>
<td>55</td>
<td>21405</td>
</tr>
<tr>
<td>suesCar</td>
<td>&quot;SUV&quot;</td>
<td>2001</td>
<td>0.45</td>
<td>35</td>
<td>9864</td>
</tr>
<tr>
<td>other</td>
<td>&quot;Sedan&quot;</td>
<td>2000</td>
<td>0.9</td>
<td>55</td>
<td>21405</td>
</tr>
</tbody>
</table>
```

Passing Arguments

- **Pass by value**
  - For primitive type, parameter initialized to value of argument in call
- **Pass by reference**
  - For a class type, formal parameter is initialized to the address of the object in the call
- **An argument cannot be changed by being passed to a method!**
  - An object referenced by an argument can be changed by a method.

Alternate implementation of `matchSpeed` *

```
* Alternate implementation of matchSpeed *

```java
public void matchSpeed(Automobile other) {
    otherCar = new Automobile(other.getModel(),
                                other.getYear(),
                                other.getFuelLevel(),
                                other.getMileage());
}
```

* WRONG!
Before calling `matchSpeed` *

```java
bobsCar = matchSpeed(suesCar);
```

- `suesCar` is associated with `other`
- `other` is a parameter, an alias of `suesCar`
- `otherCar` is a local variable with the same instance values as `other`

After executing the statement

```java
other = new Automobile( ... );
```

- `other` refers to a new memory location
- `otherCar` and the object it referenced no longer exist
- `suesCar.speed` is unchanged
- Memory allocated by `new` is garbage

After `matchSpeed` * terminates

- `other` no longer exists
- `otherCar` and the object it referenced no longer exist
- `suesCar.speed` is unchanged
- Memory allocated by `new` is garbage

Wrapper Classes

- Sometimes, you need to treat primitive types like objects
- For example, you might want to turn the integer 42 into the string “42”
- To do this, you need to:
  - Create a Wrapper class for the integer
  - Call the `toString()` method for that wrapper class
Wrapper Classes
To treat primitive types as objects, you must use wrapper classes

<table>
<thead>
<tr>
<th>Primitive Class</th>
<th>Wrapper Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>Byte</td>
</tr>
<tr>
<td>boolean</td>
<td>Boolean</td>
</tr>
<tr>
<td>char</td>
<td>Character</td>
</tr>
<tr>
<td>double</td>
<td>Double</td>
</tr>
<tr>
<td>float</td>
<td>Float</td>
</tr>
<tr>
<td>int</td>
<td>Integer</td>
</tr>
<tr>
<td>long</td>
<td>Long</td>
</tr>
<tr>
<td>short</td>
<td>Short</td>
</tr>
</tbody>
</table>

Wrapper Class Example
```java
int size = 42;
Integer sizeWrapper = new Integer(size);
String sizeString = sizeWrapper.toString();
```

Wrapper Class Functions
There are lots of other functions you can call on a wrapper class:
- `doubleValue()` – returns the value as a double
- `intValue()` – returns the value as an integer

To see the complete list, check out the Java 5.0 API

Auto-boxing
• In Java 5.0, conversion between primitive types and the corresponding wrapper classes is automatic
• This is called auto-boxing

```java
Integer i = 42; // auto-boxing
// same as Integer i = new Integer(42)
int x = i; // auto-unboxing
// same as int x = i.intValue();
```

Fancy Auto-boxing
You can even make auto-boxing work inside arithmetic expressions eg.
```java
Double e = d + 1; // d is a Double
```

Let’s look at the steps that this involves:
1. Auto-unbox d into a double
2. Add 1
3. Auto-box the result into a new Double
4. Store a reference to the newly created wrapper object in e

A Note About Wrappers
• Storing wrapper numbers is quite inefficient
• This is because you often only want to store the raw value but the wrapper class stores other stuff in addition to the value
• If you care about efficiency, use the primitive type where possible