Language Basics – Expressions, Statements, Blocks

Expressions

- **Expression**: Legal combination of constants, variables and operators
- Can be (and typically are) nested
- Expressions evaluate to a *value* of a certain *type*

Given: \( \text{int } a = 73; \) \( \text{boolean someArray[]} = \text{new boolean}[5]; \)

<table>
<thead>
<tr>
<th>Example</th>
<th>Evaluates to</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>48</td>
<td>48</td>
<td>int</td>
</tr>
<tr>
<td>2.0 / 3.0</td>
<td>0.6666666666</td>
<td>double</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>boolean</td>
</tr>
<tr>
<td>15 / 8</td>
<td>1</td>
<td>int</td>
</tr>
<tr>
<td>(17 + (3 * 9)) % 3</td>
<td>2</td>
<td>int</td>
</tr>
<tr>
<td>a + 1</td>
<td>74</td>
<td>int</td>
</tr>
<tr>
<td>( a * 9.0 / \text{someArray}.length )</td>
<td>131.4</td>
<td>double</td>
</tr>
</tbody>
</table>

Some expressions have so-called **side-effects**

Given: \( \text{int } a = 73; \) \( \text{int } b; \)

<table>
<thead>
<tr>
<th>Example</th>
<th>Value</th>
<th>Side-effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a = 84 )</td>
<td>84</td>
<td>Assign 84 to ( a )</td>
</tr>
<tr>
<td>( b = (a = 48) )</td>
<td>48</td>
<td>Assign 48 to both ( a ) and ( b )</td>
</tr>
<tr>
<td>++a</td>
<td>48</td>
<td>Assign 49 to ( a )</td>
</tr>
<tr>
<td>++a</td>
<td>50</td>
<td>Assign 50 to ( a )</td>
</tr>
<tr>
<td>new Bicycle()</td>
<td></td>
<td>Create and initialize new instance of class Bicycle in memory</td>
</tr>
<tr>
<td>new double[10]</td>
<td></td>
<td>Create and initialize new array of double</td>
</tr>
</tbody>
</table>

Reference to a new instance of Bicycle, e.g. <1150>
Statements

- **Statement**: Complete unit of execution (ends with ",")
  - **Expression statements**:
    - Assignment expressions
      - `a = (17 + (3 * 9)) % 3;`
    - Use of `++` or `--`
      - `a++;`
    - Method invocations
      - `someObject.methodOne();`
    - Object creation expressions
      - `new SomeClass();`
  - **Declaration statements**
    - `int a = 0;`

Blocks

- **Block**: Group of zero or more statements enclosed in "{'...'}"
  - `if (a == b) {
      c = 17;
      f++;
      bbb.someMethod();
    }  // end block`

Control Flow Statements

- Variables **declared** inside a block are only visible from within that block:
  - `int a = 7, b = 6;`
  - `if (a != b) {
      int c;
      c = a * b;
      System.out.println(c);
    }  // end block`

System.out.println(c);  // ERROR: c unavailable
Language Basics – Control Flow Statements

- **if** and **else** have a straightforward meaning:

```java
void applyBrakes(){
    if (speed > 0) {
        speed = speed - 1;
    }
}
```

```java
void applyBrakes(){
    if (speed > 0) {
        speed--;
    } else {
        System.err.println("The bicycle has already stopped!");
    }
}
```

- **switch**: Equivalent to sequence of chained if else statements

Language Basics – Control Flow Statements

- **while**: do **something** as long as some **condition** (boolean expression) is true

```java
int count = 1;
while (count < 8) {
    System.out.print("#" + count + " ");
    count++;
}
```

- **do while**: similar to "while", but check **condition** at the end of execution of **something** instead of at the beginning

```java
int count = 1;
do {
    System.out.print("#" + count + " ");
    count++;
} while (count < 8)
```

Language Basics – Control Flow Statements

- **for**: usually means to do **something** for a fixed number of times:

```java
for (int i=0; i<7; i++) {
    System.out.print("#" + i + " ");
}
```

- General form:

```java
for (initialization; termination; update) {
    statement
}
```

- **initialization** expression: Executed once at the beginning of first loop
- **termination** expression: If true then execute statement(s), else exit loop
- **update** expression: Executed after each iteration of the loop

Language Basics – Control Flow Statements

- for: usually means to do **something** for a fixed number of times:

```java
for (int i=0; i<7; i++) {
    System.out.print("#" + i + " ");
}
```

- General form:

```java
for (initialization; termination; update) {
    statement
}
```

- **for equivalent to while**

  ```java
  for (initialization; termination; update) {
  
  ```

  ```java
  while (termination) {
    statement* update;
  }
  ```

- **initialization** expression: Executed once at the beginning of first loop
- **termination** expression: If true then execute statement(s), else exit loop
- **update** expression: Executed after each iteration of the loop
3 Classes, Objects, Inheritance

Deepening readings:

http://java.sun.com/docs/books/tutorial/java/javaO/classes.html
http://java.sun.com/docs/books/tutorial/java/javaO/objects.html
http://java.sun.com/docs/books/tutorial/java/javaO/more.html
http://java.sun.com/docs/books/tutorial/java/landl/subclasses.html
3 Classes, Objects, Inheritance

Deepening readings:

http://java.sun.com/docs/books/tutorial/java/ann0/classes.html
http://java.sun.com/docs/books/tutorial/java/ann0/objects.html
http://java.sun.com/docs/books/tutorial/java/ann0/more.html
http://java.sun.com/docs/books/tutorial/java/land/subclasses.html

---

```java
class Bicycle {
    public int cadence = 0;
    public int speed = 0;
    public int gear = 1;

    public Bicycle(int startCadence, int startSpeed, int startGear) {
        gear = startGear;
        cadence = startCadence;
        speed = startSpeed;
    }

    public void changeCadence(int newValue) {
        cadence = newValue;
    }

    public void changeGear(int newValue) {
        gear = newValue;
    }

    public void applyPower() {
        speed += cadence;
    }
}
```

```java
public class MountainBike extends Bicycle {
    public int seatHeight;

    public MountainBike(int startHeight, int startCadence, int startSpeed, int startGear) {
        super(startCadence, startSpeed, startGear);
        seatHeight = startHeight;
    }

    public void applyPower() {
        super.applyPower();
        seatHeight += 2;
    }
}
```

---

- **Class definition (general form):**
  ```java
  modifier class MyClass extends MySuperClass
      implements YourInterface1, ..., YourInterfaceN
  
  { // fields, constructors, methods
  }
  ```

- **(Access) modifier:**
  ```java
  certain combinations of {public, protected, private, static, final}
  ```

- **Field declaration (general form):**
  ```java
  modifier type name;
  ```

- **(Access) modifier:**
  ```java
  certain combinations of {public, protected, private, static, final}
  ```

- **type:** Any primitive or reference type
### Constructor declaration (general form):
```
modifier MyClass ( parameter* ) throwsClause {
    statement*
}
```

- **Constructor parameters**: 
  - `parameter*` (later)
  - `throwsClause*` (later)
  - `statement*`: statement(s) to execute

---

### Why do we need constructors?
- Ensure complete and consistent initialization after object creation
- Access superclass constructors:
  - Construct object according to definition of superclass, then add specifics
- Provide several constructors for respective use-cases

```java
class Bicycle {
    public int cadence;
    public int speed;
    public int gear;

    public Bicycle(int c, int s, int g) {
        cadence = c;
        speed = s;
        gear = g;
    }
}
```

```java
class Tandem extends Bicycle {
    public int numberOfDrivers;

    public Tandem(int c, int s, int g, int n) {
        super(c, s, g);
        numberOfDrivers = n;
    }
}
```
Why do we need constructors?

- Ensure complete and consistent initialization after object creation
- Access superclass constructors:
  Construct object according to definition of superclass, then add specifics
- Provide several constructors for respective use-cases

```java
class Bicycle {
    public int cadence;
    public int speed;
    public int gear;

    public Bicycle(int c, int s, int g) {
        speed = s;
        gear = g;
    }
}

class Tandem extends Bicycle {
    public int numberOfDrivers;

    public Tandem(int c, int s, int g, int n) {
        super(c, s, g);
        numberOfDrivers = n;
    }
}
```