Variables

- Variables have a type
  - Primitive type
  - Reference type

<table>
<thead>
<tr>
<th>Definition</th>
<th>Declaration</th>
<th>Instantiation</th>
<th>Manipulation</th>
<th>Equality</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primitive</td>
<td>predefined</td>
<td>int a;</td>
<td>a = 117;</td>
<td>a == b;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>a = b + 42;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reference</td>
<td>class Student { // Fields and // methods ... }</td>
<td>Student heiner;</td>
<td>heiner = new Student();</td>
<td>heiner.age = 21;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>heiner.equals(sabine);</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Definition</th>
<th>Declaration</th>
<th>Instantiation</th>
<th>Manipulation</th>
<th>Equality</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primitive</td>
<td>predefined</td>
<td>int a;</td>
<td>a = 117;</td>
<td>a == b;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>a = b + 42;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reference</td>
<td>class Student { // Fields and // methods ... }</td>
<td>Student heiner;</td>
<td>heiner = new Student();</td>
<td>heiner.age = 21;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>heiner.equals(sabine);</td>
</tr>
</tbody>
</table>
**Language Basics – Variables**

**Variables**

- Variables have a type
  - Primitive type
  - Reference type

<table>
<thead>
<tr>
<th></th>
<th>Definition</th>
<th>Declaration</th>
<th>Instantiation</th>
<th>Manipulation</th>
<th>Equality</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Primitive</strong></td>
<td>predefined</td>
<td>int a;</td>
<td>a = 117;</td>
<td>a = b + 42;</td>
<td>a == b;</td>
</tr>
<tr>
<td><strong>Reference</strong></td>
<td>class Student { // Fields and // methods ... }</td>
<td>Student heiner;</td>
<td>heiner = new Student();</td>
<td>heiner.age = 21; heiner.yawn();</td>
<td>heiner.equals(sabine);</td>
</tr>
</tbody>
</table>

**Reference Type Variables**

- Reference type variables "point" to an object of the reference type

```java
bikel = new Bicycle();
bike2 = new Bicycle();

boolean c;
c = bikel.equals(bike2);
   // c == true
   c = (bikel == bike2);
   // c == false
```

<table>
<thead>
<tr>
<th>Memory (simplified model)</th>
<th>cell nr</th>
<th>cell name</th>
<th>cell content</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1149</td>
<td>bikel</td>
<td>&lt;1150&gt;</td>
<td></td>
</tr>
<tr>
<td>1150</td>
<td>bikel.cadence</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1151</td>
<td>bikel.speed</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1152</td>
<td>bikel.gear</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1327</td>
<td>bike2</td>
<td>&lt;1405&gt;</td>
<td></td>
</tr>
<tr>
<td>1405</td>
<td>bike2.cadence</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1406</td>
<td>bike2.speed</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1407</td>
<td>bike2.gear</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

**Reference Type Variables**

- Reference type variables "point" to an object of the reference type

```java
bikel = new Bicycle();
bike2 = new Bicycle();
bikel.gear = 3;

bikel = bike2;

boolean c;
c = bikel.equals(bike2);
   // c == true
   c = (bikel == bike2);
   // c == true
```

<table>
<thead>
<tr>
<th>Memory (simplified model)</th>
<th>cell nr</th>
<th>cell name</th>
<th>cell content</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1149</td>
<td>bikel</td>
<td>&lt;1405&gt;</td>
<td></td>
</tr>
<tr>
<td>1150</td>
<td>bikel.cadence</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1151</td>
<td>bikel.speed</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1152</td>
<td>bikel.gear</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1327</td>
<td>bike2</td>
<td>&lt;1405&gt;</td>
<td></td>
</tr>
<tr>
<td>1405</td>
<td>bike2.cadence</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1406</td>
<td>bike2.speed</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1407</td>
<td>bike2.gear</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Arrays

- Array: "Indexed list" of elements
- Holds a fixed number of variables of a certain type (primitive or reference)
- Is itself a reference type (see next slide)

```java
int[] someArray;
someArray = new int[6];
someArray[0] = 23;
someArray[1] = 12;
someArray[5] = 4 + someArray[2];

String[] someOtherArray;
someOtherArray = new String[30];
someOtherArray[17] = "bla bla";

AnyClass[] thirdArray;
thirdArray = new AnyClass[45];
thirdArray[44] = new AnyClass();
thirdArray[22 * 2].someMethod();
```

Arrays

- Array: "Indexed list" of elements
- Holds a fixed number of variables of a certain type (primitive or reference)
- Is itself a reference type (see next slide)

```java
int[] someArray;
someArray = new int[6];
someArray[0] = 23;
someArray[1] = 12;
someArray[5] = 4 + someArray[2];

String[] someOtherArray;
someOtherArray = new String[30];
someOtherArray[17] = "bla bla";

AnyClass[] thirdArray;
thirdArray = new AnyClass[45];
thirdArray[44] = new AnyClass();
thirdArray[22 * 2].someMethod();
```
Arrays

- Array: "Indexed list" of elements
- Holds a fixed number of variables of a certain type (primitive or reference)
- Is itself a reference type (see next slide)

```java
int[] someArray;
someArray = new int[6];
someArray[0] = 23;
someArray[1] = 12;
someArray[5] = 4 + someArray[2];

String[] someOtherArray;
someOtherArray = new String[30];
someOtherArray[17] = "bla bla";

AnyClass[] thirdArray;
thirdArray = new AnyClass[45];
thirdArray[44] = new AnyClass();
thirdArray[22 * 2].someMethod();
```

Arrays

- Array: "Indexed list" of elements
- Holds a fixed number of variables of a certain type (primitive or reference)
- Is itself a reference type (see next slide)

```java
int[] someArray;
someArray = new int[6];
someArray[0] = 23;
someArray[1] = 12;
someArray[5] = 4 + someArray[2];

String[] someOtherArray;
someOtherArray = new String[30];
someOtherArray[17] = "bla bla";

AnyClass[] thirdArray;
thirdArray = new AnyClass[45];
thirdArray[44] = new AnyClass();
thirdArray[22 * 2].someMethod();
```
Arrays

- **Array**: "Indexed list" of elements
- Holds a fixed number of variables of a certain type (primitive or reference)
- Is itself a reference type (see next slide)

```java
int[] someArray = new int[6];
someArray[0] = 23;
someArray[1] = 12;
someArray[5] = 4 + someArray[2];
String[] someOtherArray = new String[30];
someOtherArray[17] = "bla bla";
AnyClass[] thirdArray = new AnyClass[45];
thirdArray[44] = new AnyClass();
thirdArray[22 * 2].someMethod();
```

- **Array is itself a reference type**: 

```java
int[] someArray = new int[3];
int[] anotherArray = new int[3];
someArray[2] = 7;
anotherArray[1] = 8;
```

- **Length property**: 

```java
int l = someArray.length; // l == 3
```
Operators

- Operators (mostly) act on variables of primitive types. **Examples:**

  **Assignment Operator**
  \[ a = b + 1; \]
  \[ \text{bik1.copy}(); \]

  **Arithmetic Operators**
  \[ + \] Additive operator
  \[ - \] Subtraction operator
  \[ * \] Multiplication operator
  \[ / \] Division operator
  \[ \% \] Remainder operator

  \[
  \begin{align*}
  \text{int } a & = 1 + 1; \\
  \text{float } f & = 10.0f - 23.0f; \\
  \text{double } d & = z * z; \\
  \text{int } a & = 17 \div 9 \\
  \text{int } a & = 17 \mod 9
  \end{align*}
  \]

  **Unary Operators**
  \[ + \] Unary plus operator; (not very useful)
  \[ - \] Unary minus operator; negates an expression
  \[ ++ \] Increment by 1
  \[ -- \] Decrement by 1
  \[ ! \] Inverse value of a boolean

  \[
  \begin{align*}
  \text{int } a & = -1; \\
  \text{int } b & = +a; \\
  \text{int } a & = -1; \\
  \text{int } b & = +a; \\
  \text{int } a & = 0; \\
  \text{int } a & = 0; \\
  \text{int } a & = 1; \\
  \text{int } a & = -1; \\
  \end{align*}
  \]

- Operators (mostly) act on variables of primitive types. **Examples:**

  **Assignment Operator**
  \[ a = b + 1; \]
  \[ \text{bik1.copy}(); \]

  **Arithmetic Operators**
  \[ + \] Additive operator
  \[ - \] Subtraction operator
  \[ * \] Multiplication operator
  \[ / \] Division operator
  \[ \% \] Remainder operator

  \[
  \begin{align*}
  \text{int } a & = 1 + 1; \\
  \text{float } f & = 10.0f - 23.0f; \\
  \text{double } d & = z * z; \\
  \text{int } a & = 17 \div 9 \\
  \text{int } a & = 17 \mod 9
  \end{align*}
  \]

  **Unary Operators**
  \[ + \] Unary plus operator; (not very useful)
  \[ - \] Unary minus operator; negates an expression
  \[ ++ \] Increment by 1
  \[ -- \] Decrement by 1
  \[ ! \] Inverse value of a boolean

  \[
  \begin{align*}
  \text{int } a & = -1; \\
  \text{int } b & = +a; \\
  \text{int } a & = -1; \\
  \text{int } b & = +a; \\
  \text{int } a & = 0; \\
  \text{int } a & = 0; \\
  \text{int } a & = 1; \\
  \text{int } a & = -1; \\
  \end{align*}
  \]
Operators

- **Operators (mostly) act on variables of primitive types.**  **Examples:**

  **Assignment Operator**
  
  - Simple assignment operator (also for reference types)
  
  ```
  a = b+1; bike2 = bike1.copy();
  ```

  **Arithmetic Operators**
  
  - Additive operator
  
  ```
  double aaa = b + 1.7f; int a = l + l;
  ```
  
  - Subtraction operator
  
  ```
  int b = c - 9; float f = 10.0f - 23.0f;
  ```
  
  - Multiplication operator
  
  ```
  fd = fd * 0.1f; double d = z * z;
  ```
  
  - Division operator
  
  ```
  int a = 17 / 9 // a == 1;
  float eee = 13.0f / 2.0f // eee == 6.5f;
  ```

  - Remainder operator
  
  ```
  int a = 17 % 9 // a == 9;
  ```

  **Unary Operators**
  
  - Unary plus operator; (not very useful)
  
  ```
  int a = -l; int b = +a; // b == -l
  ```
  
  - Unary minus operator; negates an expression
  
  ```
  int a = -l; int b = -a; // b == -l
  ```
  
  - Increment by 1
  
  ```
  int a = 0; a++; // a == 1;
  ```
  
  - Decrement by 1
  
  ```
  int a = l; a--; // a == 0;
  ```
  
  - Inverse value of a boolean
  
  ```
  boolean b = true; c = !b; // c==false;
  ```

---

Operators

- **Operators (mostly) act on variables of primitive types.**  **Examples:**

  **Assignment Operator**
  
  - Simple assignment operator (also for reference types)
  
  ```
  a = b+1; bike2 = bike1.copy();
  ```

  **Arithmetic Operators**
  
  - Additive operator
  
  ```
  double aaa = b + 1.7f; int a = l + l;
  ```
  
  - Subtraction operator
  
  ```
  int b = c - 9; float f = 10.0f - 23.0f;
  ```
  
  - Multiplication operator
  
  ```
  fd = fd * 0.1f; double d = z * z;
  ```
  
  - Division operator
  
  ```
  int a = 17 / 9 // a == 1;
  float eee = 13.0f / 2.0f // eee == 6.5f;
  ```

  - Remainder operator
  
  ```
  int a = 17 % 9 // a == 9;
  ```

  **Unary Operators**
  
  - Unary plus operator; (not very useful)
  
  ```
  int a = -l; int b = +a; // b == -l
  ```
  
  - Unary minus operator; negates an expression
  
  ```
  int a = -l; int b = -a; // b == -l
  ```
  
  - Increment by 1
  
  ```
  int a = 0; a++; // a == 1;
  ```
  
  - Decrement by 1
  
  ```
  int a = l; a--; // a == 0;
  ```
  
  - Inverse value of a boolean
  
  ```
  boolean b = true; c = !b; // c==false;
  ```
Operators (mostly) act on variables of primitive types. Examples:

### Assignment Operator

Simple assignment operator (also for reference types)
```
a = b + 1; bike2 = bike1.copy();
```

### Arithmetic Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Additive operator</td>
<td>double a = b + 1.7; int a = 1 + 1;</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction operator</td>
<td>int b = c - 9; float f = 10.0f - 23.0f;</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication operator</td>
<td>int b = c * 9; double d = z * z;</td>
</tr>
<tr>
<td>/</td>
<td>Division operator</td>
<td>int a = 17 / 9 // a == 1</td>
</tr>
<tr>
<td>%</td>
<td>Remainder operator</td>
<td>int a = 17 % 9 // a == 8</td>
</tr>
</tbody>
</table>

### Unary Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Unary plus operator; (not very useful)</td>
<td>int a = -1; int b = +a; // b == -1</td>
</tr>
<tr>
<td>-</td>
<td>Unary minus operator; negates an expression</td>
<td>int a = -1; int b = -a; // b == 1</td>
</tr>
<tr>
<td>++</td>
<td>Increment by 1</td>
<td>int a = 0; a++; // a == 1</td>
</tr>
<tr>
<td>--</td>
<td>Decrement by 1</td>
<td>int a = 1; a--; // a == 0</td>
</tr>
<tr>
<td>!</td>
<td>Inverse value of a boolean</td>
<td>boolean b = true; c = !b; // c==false</td>
</tr>
</tbody>
</table>

### Equality and Relational Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>==</td>
<td>Equal to</td>
<td>boolean a = (1 == 1); // a == true</td>
</tr>
<tr>
<td>!=</td>
<td>Not equal to</td>
<td>boolean a = (1 != 1); // a == false</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
<td>boolean a = (17 &gt; 12); // a == true</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or equal to</td>
<td>boolean a = (17 &gt; 12); // a == true</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
<td>boolean a = (17 &gt; 12); // a == true</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal to</td>
<td>boolean a = (17 &gt; 12); // a == true</td>
</tr>
</tbody>
</table>

### Conditional Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp; &amp;</td>
<td>Conditional-AND</td>
<td>a = false; b = true; c = a &amp;&amp; b; // c == false</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>?:</td>
<td>Ternary (shorthand for if-then-else statement, use if-then-else instead!)</td>
<td>boolean a = 1 ? 2 : 3; // a == 2</td>
</tr>
</tbody>
</table>

### Reference Type Comparison Operator

```
instanceof
```

Compares an object to a specified type
```
instanceof Vector; // b== true;
```

### Bitwise and Bit Shift Operators

(not that important for us; see URL below)

http://docs.oracle.com/javase/tutorial/java/nutsandbolts/op2.html
Language Basics – Operators

Equality and Relational Operators

=== Equal to
!= Not equal to
> Greater than
>= Greater than or equal to
c.< Less than
<= Less than or equal to

Conditional Operators

&& Conditional-AND
|| Conditional-OR
?: Ternary (shorthand for if-then-else statement, use if-then-else instead)

Reference Type Comparison Operator

instanceof Compares an object to a specified type

Bitwise and Bit Shift Operators

(not that important for us; see URL below)

Vector z = new Vector();
boolean b = z instanceof Vector;
// b == true;

http://docs.oracle.com/javase/tutorial/java/nutsandbolts/op3.html

Equality and Relational Operators

=== Equal to
!= Not equal to
> Greater than
>= Greater than or equal to
c.< Less than
<= Less than or equal to

Conditional Operators

&& Conditional-AND
|| Conditional-OR
?: Ternary (shorthand for if-then-else statement, use if-then-else instead)

Reference Type Comparison Operator

instanceof Compares an object to a specified type

Bitwise and Bit Shift Operators

(not that important for us; see URL below)

Vector z = new Vector();
boolean b = z instanceof Vector;
// b == true;

http://docs.oracle.com/javase/tutorial/java/nutsandbolts/op3.html
### Equality and Relational Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Code Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>==</code></td>
<td>Equal to</td>
<td>boolean a = (1 == 1); // a == true</td>
</tr>
<tr>
<td><code>!=</code></td>
<td>Not equal to</td>
<td>boolean a = (1 != 1); // a == false</td>
</tr>
<tr>
<td><code>&gt;</code></td>
<td>Greater than</td>
<td>boolean a = (17 &gt; 12); // a == true, etc.</td>
</tr>
<tr>
<td><code>&gt;=</code></td>
<td>Greater than or equal to</td>
<td>etc.</td>
</tr>
<tr>
<td><code>&lt;</code></td>
<td>Less than</td>
<td>etc.</td>
</tr>
<tr>
<td><code>&lt;=</code></td>
<td>Less than or equal to</td>
<td>etc.</td>
</tr>
</tbody>
</table>

### Conditional Operators

<table>
<thead>
<tr>
<th>Boolean Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&amp;&amp;</code></td>
<td>Conditional-AND</td>
</tr>
<tr>
<td>`</td>
<td></td>
</tr>
<tr>
<td><code>? :</code></td>
<td>Ternary (short-hand for if-then-else statement, use if-then-else instead!)</td>
</tr>
</tbody>
</table>

### Reference Type Comparison Operator

<table>
<thead>
<tr>
<th>Comparison Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>instanceof</code></td>
<td>Compares an object to a specified type</td>
</tr>
</tbody>
</table>

### Bitwise and Bit Shift Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&amp;</code></td>
<td>Bitwise AND</td>
</tr>
<tr>
<td>`</td>
<td>`</td>
</tr>
<tr>
<td><code>^</code></td>
<td>Bitwise XOR</td>
</tr>
<tr>
<td><code>&lt;&lt;</code></td>
<td>Bitwise left shift</td>
</tr>
<tr>
<td><code>&gt;&gt;</code></td>
<td>Bitwise right shift</td>
</tr>
</tbody>
</table>

### Operators (mostly) act on variables of primitive types. Examples:

#### Assignment Operator

<table>
<thead>
<tr>
<th>Description</th>
<th>Code Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple assignment operator</td>
<td><code>a = b+1; bike2 = bike1.copy();</code></td>
</tr>
</tbody>
</table>

#### Arithmetic Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Code Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>+</code></td>
<td>Additive operator</td>
<td>double aaa = b + 1.7; int a = l + 1;</td>
</tr>
<tr>
<td><code>-</code></td>
<td>Subtraction operator</td>
<td>int b = c - 9; float f = -10.0f - 23.0f;</td>
</tr>
<tr>
<td><code>*</code></td>
<td>Multiplication operator</td>
<td>fd = fd * 0.1f; double d = z * 2;</td>
</tr>
<tr>
<td><code>/</code></td>
<td>Division operator</td>
<td>int a = 17 / 9; float swa = 13.0f / 2.0f;</td>
</tr>
<tr>
<td><code>%</code></td>
<td>Remainder operator</td>
<td>int a = 17 % 9;</td>
</tr>
</tbody>
</table>

#### Unary Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>+</code></td>
<td>Unary plus operator (not very useful)</td>
</tr>
<tr>
<td><code>-</code></td>
<td>Unary minus operator; negates an expression</td>
</tr>
<tr>
<td><code>++</code></td>
<td>Increment by 1</td>
</tr>
<tr>
<td><code>--</code></td>
<td>Decrement by 1</td>
</tr>
<tr>
<td><code>!</code></td>
<td>Inverse value of a boolean</td>
</tr>
</tbody>
</table>

```java
boolean b = true; c = !b; // c==false
```
### Equality and Relational Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>==</td>
<td>Equal to</td>
<td>boolean a = (1 == 1); // a == true</td>
</tr>
<tr>
<td>!=</td>
<td>Not equal to</td>
<td>boolean a = (1 != 1); // a == false</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
<td>boolean a = (17 &gt; 12); // a == true;</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or equal to</td>
<td>etc.</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
<td>etc.</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal to</td>
<td>etc.</td>
</tr>
</tbody>
</table>

### Conditional Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;&amp;</td>
<td>Conditional-AND</td>
<td>a = false; b = true; c = a &amp;&amp; b; // c == false;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>? :</td>
<td>Ternary (shorthand for if-then-else statement, use if-then-else instead!)</td>
<td>etc.</td>
</tr>
</tbody>
</table>

### Reference Type Comparison Operator

<table>
<thead>
<tr>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>instanceof</td>
<td>Vector z = new Vector(); boolean b = z instanceof Vector; // b == true;</td>
</tr>
</tbody>
</table>

### Bitwise and Bit Shift Operators

- **(not that important for us; see URL below)**
- **(not that important for us; see URL below)**

---

### There is a **fixed precedence** of operators

### Simple: Use brackets "(" ... ")" to enforce precedence as desired!

```java
int a = ((7 + 4) * 8) % 3; // a == 1
```

---

### Important: Dereference operator for reference types: dot-operator "."
**Language Basics – Operators**

- There is a **fixed precedence** of operators
- Simple: Use brackets "( ... )" to enforce precedence as desired!

```java
int a = ((7 + 4) * 8) % 3;  // a == 1
```

- Important: Dereference operator for reference types: dot-operator "." 

```java
String s1 = s1.concatenate(s2);
bike1.cadence = 4;
bike1.changeGear(5);
```

**Language Basics – Operators**

- There is a **fixed precedence** of operators
- Simple: Use brackets "( ... )" to enforce precedence as desired!

```java
int a = ((7 + 4) * 8) % 3;  // a == 1
```

- Important: Dereference operator for reference types: dot-operator "." 

```java
String s1 = s1.concatenate(s2);
bike1.cadence = 4;
bike1.changeGear(5);
```

---

**Language Basics – Expressions, Statements, Blocks**

**Expressions**

- Some expressions have so-called **side-effects** (in most cases the only important aspect about the expression!!!)

Given:  `int a = 73; 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 (l)</td>
</tr>
<tr>
<td>++a</td>
<td>50</td>
<td>Assign 50 to a (l)</td>
</tr>
<tr>
<td>new Bicycle()</td>
<td>Reference to the new instance of Bicycle, e.g. &lt;1150&gt;</td>
<td>Create and initialize new instance of class Bicycle in memory</td>
</tr>
<tr>
<td>new double[10]</td>
<td>Reference to the new array of double</td>
<td>Create and initialize new array in memory</td>
</tr>
</tbody>
</table>
Expressions

- **Expression**: Legal combination of values, operators, and operands.
- Can be (and typically are) nested.
- Expressions evaluate to a value.

**Example**

- `48` type: `int`
- `2.0 / 3.0` type: `double`
- `true` type: `boolean`
- `15 / 8` type: `int`
- `17 + (3 * 9)` type: `int`
- `a + 1` type: `int`
- `a * 9.0 / someArray.length` type: `double`

Expressions

- Some expressions have so-called **side-effects** (in most cases the only important aspect about the expression!!!)

**Example**

<table>
<thead>
<tr>
<th>Example</th>
<th>Value</th>
<th>Side-effect</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a = 84</code></td>
<td>84</td>
<td>Assign 84 to <code>a</code></td>
</tr>
<tr>
<td><code>b = (a = 48)</code></td>
<td>48</td>
<td>Assign 48 to both <code>a</code> and <code>b</code></td>
</tr>
<tr>
<td><code>a++</code></td>
<td>48</td>
<td>Assign 49 to <code>a</code> (l)</td>
</tr>
<tr>
<td><code>++a</code></td>
<td>50</td>
<td>Assign 50 to <code>a</code> (l)</td>
</tr>
<tr>
<td><code>new Bicycle()</code></td>
<td></td>
<td>Reference to the new instance of Bicycle, e.g. <code>&lt;1150&gt;</code> Create and initialize new instance of class Bicycle in memory</td>
</tr>
<tr>
<td><code>new double[10]</code></td>
<td></td>
<td>Reference to the new array of double Create and initialize new array in memory</td>
</tr>
</tbody>
</table>
### Statements

- **Statement:** Complete unit of execution (ends with `;`)
- **Expression statements:**
  - Assignment expressions: \( a = (17 + (3 * 9)) \% 3; \)
  - Use of `++` or `--`: \( a++; \)
  - Method invocations: \( \text{someObject.methodOne();} \)
  - Object creation expressions: \( \text{new SomeClass();} \)
- **Declaration statements:** \( \text{int a;} \)
- **Blocks**
  - (next slide)
- **Control flow statements**
  - (later)

### Blocks

- Variables declared inside a block are only visible from within that block:

```java
int a = 7, b = 6;

if (a != b) {
    // begin block
    int c;
    c = a * b;
    System.out.println(c);
} // end block
System.out.println(c); // ERROR: c unavailable
```

### Control Flow Statements

- **Control flow statements:**
  - Allow for deviation of control flow from sequential order of statements:
    - **Conditionals:** if, if else, switch
    - **Loops:** while, do while, for
    - **Branches:** break, continue, return
Control Flow Statements

- **Control flow statements:** Allow for deviation of control flow from sequential order of statements:
  - **Conditionals:** if, if else, switch
  - **Loops:** while, do while, for
  - **Branches:** break, continue, return

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

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

```java
void applyBrakes()
    if (speed > 10) {
        speed = speed - 2;  // break really hard
    } else if (speed > 0) {
        speed--;            // soft brakes
    } else {
        System.err.println("The bicycle has already stopped!");
    }
}
```

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

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

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

```java
void applyBrakes()
    if (speed > 10) {
        speed = speed - 2;  // break really hard
    } else if (speed > 0) {
        speed--;            // soft brakes
    } else {
        System.err.println("The bicycle has already stopped!");
    }
}
```

- **switch:** Equivalent to sequence of chained if else statements
• if and if else have a straightforward meaning:

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

```java
void applyBrakes()
{
    if (speed > 10) {
        speed = speed - 2; // break really hard
    } else if (speed > 0) {
        speed--; // soft brakes
    } else {
        System.err.println("The bicycle has already stopped!");
    }
}
```

• switch: Equivalent to sequence of chained if else statements

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

```java
void applyBrakes()
{
    if (speed > 10) {
        speed = speed - 2; // break really hard
    } else if (speed > 0) {
        speed--; // soft brakes
    } else {
        System.err.println("The bicycle has already stopped!");
    }
}
```

• switch: Equivalent to sequence of chained if else statements
• if and if else have a straightforward meaning:

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

void applyBrakes()
{
    if (speed > 10) {
        speed = speed - 2; // break really hard
    } else if (speed > 0) {
        speed--; // soft brakes
    } else {
        System.err.println(
            "The bicycle has already stopped!");
    }
}
void applyBrakes()
{
    if (speed > 10) {
        speed = speed - 2; // break really hard
    } else if (speed > 0) {
        speed--; // soft brakes
    } else {
        System.err.println(
            "The bicycle has already stopped!");
    }
}

• switch: Equivalent to sequence of chained if else statements

• if and if else have a straightforward meaning:

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

void applyBrakes()
{
    if (speed > 10) {
        speed = speed - 2; // break really hard
    } else if (speed > 0) {
        speed--; // soft brakes
    } else {
        System.err.println(
            "The bicycle has already stopped!");
    }
}
void applyBrakes()
{
    if (speed > 10) {
        speed = speed - 2; // break really hard
    } else if (speed > 0) {
        speed--; // soft brakes
    } else {
        System.err.println(
            "The bicycle has already stopped!");
    }
}

• switch: Equivalent to sequence of chained if else statements
- if and if else have a straightforward meaning:

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

void applyBrakes()
{
    if (speed > 10) {
        speed = speed - 2;  // break really hard
    } else if (speed > 0) {
        speed--;           // soft brakes
    } else {
        System.err.println(  
            "The bicycle has already stopped!");
    }
}
```

- switch: Equivalent to sequence of chained if else statements

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

void applyBrakes()
{
    if (speed > 10) {
        speed = speed - 2;  // break really hard
    } else if (speed > 0) {
        speed--;           // soft brakes
    } else {
        System.err.println(  
            "The bicycle has already stopped!");
    }
}
```

- switch: Equivalent to sequence of chained if else 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++;
}
```

output will be:  #:1 #:2 #:3 #:4 #:5 #:6 #:7

**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);
```

output will be:  #:1 #:2 #:3 #:4 #:5 #:6 #:7

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

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

output will be:  #:0 #:1 #:2 #:3 #:4 #:5 #:6

General form:

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

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

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

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

output will be:  #:1 #:2 #:3 #:4 #:5 #:6 #:7

**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);
```

output will be:  #:1 #:2 #:3 #:4 #:5 #:6 #:7
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++) { // loop will be executed 7 times
    System.out.print("\#": + i + " ");
}
```

> output will be: \#0 \#1 \#2 \#3 \#4 \#5 \#6

• 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++) { // loop will be executed 7 times
    System.out.print("\#": + i + " ");
}
```

> output will be: \#0 \#1 \#2 \#3 \#4 \#5 \#6

• 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++) { // loop will be executed 7 times
    System.out.print("#: " + i + " ");
}
```

output will be: 0:1 2:3 4:5 6:7

• **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++) { // loop will be executed 7 times
    System.out.print("#: " + i + " ");
}
```

output will be: 0:1 2:3 4:5 6:7

• **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 < 10; i++) {
    if (i == 8) {
        break;
    } else if (i % 2 == 0) {
        continue;
    }
    System.out.print("#: " + i + " ");
}
```

output will be: 1:3 5:7

• **break**: force termination of a loop
• **continue**: skip current iteration of a loop

... can be avoided in almost all relevant cases

---

Language Basics – Control Flow Statements

• **return**: terminate current method and return control flow to where the method was invoked from (will be covered shortly in more detail)
**Language Basics – Control Flow Statements**

- **break**: force termination of a loop
  
- **continue**: skip current iteration of a loop  
  
```java
for (int i = 0; i < 10; i++) {
    if (i == 8) {
        break;
    } else if (i % 2 == 0) {
        continue;
    }
    System.out.print("#: " + i + " ");
}
```

> output will be: #:1 #:3 #:5 #:7

- **return**: terminate current method and return control flow to where the method was invoked from (will be covered shortly in more detail)

---

**Language Basics – Control Flow Statements**

**Control Flow Statements**

- **Control flow statements**: Allow for deviation of control flow from sequential order of statements:
  
  - **conditionals**: if, if else, switch
  
  - **loops**: while, do while, for
  
  - **branches**: break, continue, return