Module 1: Types and Expressions

Ron K. Cytron

Department of Computer Science and Engineering
Washington University in Saint Louis
Thanks to Alan Waldman for comments that improved these slides

Prepared for 2u
Semester Online

Copyright Ron K. Cytron 2013
1.0 Introduction

- What is a type?
  - A classification of items in the world based on
    - The kinds of things an item can do
    - The
      - integers, real numbers, character strings, .....  
- As types, consider integers and real numbers
  - integer
    - Useful for counting
    - Number of playing cards in a deck
  - Real number
    - Can include fractional values
    - Temperature readings and averages, interest rates
    - On the computer, we can only approximate real numbers

- Data types in computer science
  - Represent values of interest
  - Allow operations that make sense

- Built in data types (in this module)
  - For representing integers, interest rates, names, addresses, facts

- Later, we design our own types
  - Bank account, hockey player, song

- The best choice of data type is not always obvious
  - Should you use an integer or a real number?
  - You might start with one, and change your mind later
    - Height – might be integer number of centimeters
    - But if we're measuring something small, fractional values may be needed

- There are rarely right or wrong choices
  - But a choice will have its advantages and disadvantages
  - Picking a suitable data type is an engineering process and is based on currently understood constraints and best planning for the future. The future can at times seem uncertain.
  - Example: Y2K

Monolog 1
End of Monologue
1.1 Our first data type: int

- The int data type allows us to represent integer-valued items

- How many students are in this class?
- How many exams will be given?
- How many words are in this sentence?

- The above are 0 or positive, but negative values may be of interest as well
  - How many days until your 21st birthday?
  - The week before, 7
  - The week after, -7

- How has the class size changed since yesterday?
  - +5 students were added
  - -3 students dropped
1.1 Our first data type: int

- The int data type allows us to represent integer-valued items
  - How many students are in this class?
  - How many exams will be given?
  - How many words are in this sentence?
1.1 Our first data type: **int**

- The **int** data type allows us to represent integer-valued items
  - How many students are in this class?
  - How many exams will be given?
  - How many words are in this sentence?

- The above are 0 or positive, but negative values may be of interest as well
  - How many days until your 21\textsuperscript{st} birthday?
  - The week before, 7
  - The week after, -7
  - How has the class size changed since yesterday?
    - +5 students were added
    - -3 students dropped
• The int data type allows us to represent integer-valued items
  – How many students are in this class?
  – How many exams will be given?
  – How many words are in this sentence?

• The above are 0 or positive, but negative values may be of interest as well
  – How many days until your 21\textsuperscript{st} birthday?
    • The week before, 7
    • The week after, -7
  – How has the class size changed since yesterday?
    • +5 students were added
    • -3 students dropped
1.1 Our first data type: int

- Most programming languages have practical limits on the values represented by a simple data type
  - For example, Java's int type:
    - Minimum value: -2,147,483,648
    - Maximum value: 2,147,483,647
  - These seem just fine for the quantities suggested thus far
    - But how about the distance from Earth to Sun?
      - In meters: 149,597,870,700   Too big!
      - We can represent values outside of the range of an int using techniques we learn later.
      - For now, we consider an int sufficient.
  - There are things we cannot represent conveniently with an int
    - Your name (need the alphabet, not just 0—9)
    - Your height in meters (need fractions)
    - Whether you are vegetarian or not
  - We will see other data types soon that help us with those
1.1 Our first data type: int

- Most programming languages have practical limits on the values represented by a simple data type
- For example, Java’s int type:
  - Minimum value: -2,147,483,648
  - Maximum value: 2,147,483,647
1.1 Our first data type: int

- Most programming languages have practical limits on the values represented by a simple data type
- For example, Java’s int type:
  - Minimum value: -2,147,483,648
  - Maximum value: 2,147,483,647
- These seem just fine for the quantities suggested thus far
  - But how about the distance from Earth to Sun?
1.1 Our first data type: int

• Most programming languages have practical limits on the values represented by a simple data type
• For example, Java’s int type:
  – Minimum value: -2,147,483,648
  – Maximum value: 2,147,483,647
• These seem just fine for the quantities suggested thus far
  – But how about the distance from Earth to Sun?
    • In meters: 149,597,870,700 Too big!
  – We can represent values outside of the range of an int using techniques we learn later.
  – For now, we consider an int sufficient.
1.1 Our first data type: int

• Most programming languages have practical limits on the values represented by a simple data type
• For example, Java’s int type:
  – Minimum value: -2,147,483,648
  – Maximum value: 2,147,483,647
• These seem just fine for the quantities suggested thus far
  – But how about the distance from Earth to Sun?
    • In meters: 149,597,870,700 Too big!
    – We can represent values outside of the range of an int using techniques we learn later.
    – For now, we consider an int sufficient.
• There are things we cannot represent conveniently with an int
  – Your name (need the alphabet, not just 0—9)
  – Your height in meters (need fractions)
  – Whether you are vegetarian or not
• We will see other data types soon that help us with those
1.1 Our first data type: \texttt{int}

• How do we represent a concept using the \texttt{int} data type? We do this by specifying:
  – The type of the concept (\texttt{int})
  – A name for the concept (numStudents, numExams, etc.)
  – A value for the concept

  • This value can change later, but it is important to have an \textit{initial} value for the concept.

\begin{verbatim}
int numStudents = 390;
\end{verbatim}
1.1 Our first data type: int

- How do we represent a concept using the int data type? We do this by specifying:
  - The type of the concept (int)
  - A name for the concept (numStudents, numExams, etc.)
  - A value for the concept
    - This value can change later, but it is important to have an *initial* value for the concept.

```java
int numStudents = 390;
```
1.1 Our first data type: int

• How do we represent a concept using the int data type? We do this by specifying:
  – The type of the concept (int)
  – A name for the concept (numStudents, numExams, etc.)
  – A value for the concept
    • This value can change later, but it is important to have an initial value for the concept.

```plaintext
int numStudents = 390;
```
1.1 Our first data type: int

• How do we represent a concept using the int data type? We do this by specifying:
  – The type of the concept (int)
  – A name for the concept (numStudents, numExams, etc.)
  – A value for the concept
    • This value can change later, but it is important to have an initial value for the concept.

    ```
    int numStudents = 390;
    ```

• There are conventions about how to pick variable names:
  – The first letter of the first word in a name begins with a lower-case letter
  – Each subsequent word begins with an upper-case letter
  – There are no spaces (blanks) in a variable name
    • This value can change later, but it is important to have an initial value for the concept.

• The name should be meaningful, clear, and relevant to the concept
1.1 Our first data type: int

- How do we represent a concept using the int data type? We do this by specifying:
  - The type of the concept (int)
  - A name for the concept (numStudents, numExams, etc.)
  - A value for the concept

- This value can change later, but it is important to have an initial value for the concept.

\[
\text{int } \textbf{numStudents} \ = \ 390;
\]

- There are conventions about how to pick variable names:
1.1 Our first data type: int

• How do we represent a concept using the int data type? We do this by specifying:
  – The type of the concept (int)
  – A name for the concept (numStudents, numExams, etc.)
  – A value for the concept
    • This value can change later, but it is important to have an initial value for the concept.

```c
int numStudents = 390;
```

• There are conventions about how to pick variable names:
  – The first letter of the first word in a name begins with a lower-case letter
  – Each subsequent word begins with an upper-case letter
  – There are no spaces (blanks) in a variable name
1.1 Our first data type: int

- How do we represent a concept using the int data type? We do this by specifying:
  - The type of the concept (int)
  - A name for the concept (numStudents, numExams, etc.)
  - A value for the concept
    - This value can change later, but it is important to have an initial value for the concept.

```c
int numStudents = 390;
```

- There are conventions about how to pick variable names:
  - The first letter of the first word in a name begins with a lower-case letter
  - Each subsequent word begins with an upper-case letter
  - There are no spaces (blanks) in a variable name

- The name should be meaningful, clear, and relevant to the concept
1.1 Our first data type: int

• How do we represent a concept using the int data type? We do this by specifying:
  – The type of the concept (int)
  – A name for the concept (numStudents, numExams, etc.)
  – A value for the concept
    • This value can change later, but it is important to have an initial value for the concept.

```java
int numStudents = 390;
```

• Any valid expression can appear on the right-hand side
1.1 Our first data type: int

• How do we represent a concept using the int data type? We do this by specifying:
  – The type of the concept (int)
  – A name for the concept (numStudents, numExams, etc.)
  – A value for the concept
    • This value can change later, but it is important to have an initial value for the concept.

    \[
    \text{int numStudents} = 390;
    \]

• An assignment statement is evaluated as follows
  – The value of the right-hand side is computed (390 in this case)
1.1 Our first data type: int

• How do we represent a concept using the int data type? We do this by specifying:
  – The type of the concept (int)
  – A name for the concept (numStudents, numExams, etc.)
  – A value for the concept
    • This value can change later, but it is important to have an initial value for the concept.

```plaintext
int numStudents = 390;
```

• An assignment statement is evaluated as follows
  – The value of the right-hand side is computed (390 in this case)
  – The result is stored at the name declared on the left-hand side
1.1 Our first data type: int

• How do we represent a concept using the int data type? We do this by specifying:
  – The type of the concept (int)
  – A name for the concept (numStudents, numExams, etc.)
  – A value for the concept
    • This value can change later, but it is important to have an initial value for the concept.

```c
int numStudents = 390;
int numWaitListed = 20;
```
1.1 Our first data type: int

• How do we represent a concept using the int data type? We do this by specifying:
  – The type of the concept (int)
  – A name for the concept (numStudents, numExams, etc.)
  – A value for the concept
    • This value can change later, but it is important to have an
      initial value for the concept.

```java
int numStudents = 390;
int numWaitListed = 20;
```

• The computer executes your statements in sequence
  – One after the other
  – From the top, reading down
1.1 Our first data type: int

• The computer executes your statements in sequence
  – One after the other
  – From the top, reading down

```java
int numStudents = 390;
in int numWaitListed = 20;
```
1.1 Our first data type: int

- The computer executes your statements in sequence
  - One after the other
  - From the top, reading down

```java
int numStudents = 390;
int numWaitListed = 20;
int totalStudents = numStudents + numWaitListed;
```

```
390 + 20

410
```
1.1 Our first data type: int

- The computer executes your statements in sequence
  - One after the other
  - From the top, reading down

```c
int numStudents = 390;
int numWaitListed = 20;
int totalStudents = numStudents + numWaitListed;
```

The diagram shows the execution of the code:

- `numStudents` is set to 390
- `numWaitListed` is set to 20
- The total is calculated as `numStudents + numWaitListed` = 410
1.1 Our first data type: int

- The computer executes your statements in sequence
  - One after the other
  - From the top, reading down

```java
int numStudents = 390;
int numWaitListed = 20;
int totalStudents = numStudents + numWaitListed;
```

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>numStudents</td>
<td>390</td>
</tr>
<tr>
<td>numWaitListed</td>
<td>20</td>
</tr>
<tr>
<td>totalStudents</td>
<td>410</td>
</tr>
</tbody>
</table>
1.1 Our first data type: int

- The computer executes your statements in sequence:
  - One after the other
  - From the top, reading down

```java
int numStudents = 390;
int numWaitListed = 20;
int totalStudents = numStudents + numWaitListed;
numStudents = numStudents + 5;
```

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>numStudents</td>
<td>390</td>
</tr>
<tr>
<td>numWaitListed</td>
<td>20</td>
</tr>
<tr>
<td>totalStudents</td>
<td>410</td>
</tr>
</tbody>
</table>
1.1 Our first data type: int

- The computer executes your statements in sequence
  - One after the other
  - From the top, reading down

```c
int numStudents = 390;
int numWaitListed = 20;
int totalStudents = numStudents + numWaitListed;
int numStudents = numStudents + 5;
```

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>numStudents</td>
<td>390</td>
</tr>
<tr>
<td>numWaitListed</td>
<td>20</td>
</tr>
<tr>
<td>totalStudents</td>
<td>410</td>
</tr>
</tbody>
</table>
1.1 Our first data type: int

- The computer executes your statements in sequence
  - One after the other
  - From the top, reading down

```
int numStudents = 390;
int numWaitListed = 20;
int totalStudents = numStudents + numWaitListed;
int numStudents = numStudents + 5;
```

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>numStudents</td>
<td>390</td>
</tr>
<tr>
<td>numWaitListed</td>
<td>20</td>
</tr>
<tr>
<td>totalStudents</td>
<td>410</td>
</tr>
</tbody>
</table>

Because we are assigning a previously-declared variable `numStudents` a new value, its type has already been declared as `int`, and we are not allowed to declare its type ever again.
1.1 Our first data type: int

• The computer executes your statements in sequence
  – One after the other
  – From the top, reading down

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>numStudents</td>
<td>390</td>
</tr>
<tr>
<td>numWaitListed</td>
<td>20</td>
</tr>
<tr>
<td>totalStudents</td>
<td>410</td>
</tr>
</tbody>
</table>

```java
int numStudents = 390;
int numWaitListed = 20;
int totalStudents = numStudents + numWaitListed;
numStudents = numStudents + 5;
```

Because we are assigning a previously-declared variable numStudents a new value, its type has already been declared as int, and we are not allowed to declare its type ever again.
1.1 Our first data type: int

- The computer executes your statements in sequence
  - One after the other
  - From the top, reading down

```java
int numStudents = 390;
int numWaitListed = 20;
int totalStudents = numStudents + numWaitListed;
```

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>numStudents</td>
<td>390</td>
</tr>
<tr>
<td>numWaitListed</td>
<td>20</td>
</tr>
<tr>
<td>totalStudents</td>
<td>410</td>
</tr>
</tbody>
</table>

```
numStudents = numStudents + 5;
```

```
390 + 5 = 395
```
1.1 Our first data type: int

- The computer executes your statements in sequence
  - One after the other
  - From the top, reading down

```
int numStudents = 390;
int numWaitListed = 20;
int totalStudents = numStudents + numWaitListed;
```

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>numStudents</td>
<td>395</td>
</tr>
<tr>
<td>numWaitListed</td>
<td>20</td>
</tr>
<tr>
<td>totalStudents</td>
<td>410</td>
</tr>
</tbody>
</table>

```
numStudents = numStudents + 5;
```

```
\[ 390 + 5 \rightarrow 395 \]
```
1.1 Our first data type: int

• The computer executes your statements in sequence
  – One after the other
  – From the top, reading down

```
int numStudents = 390;
int numWaitListed = 20;
int totalStudents = numStudents + numWaitListed;
```

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>numStudents</td>
<td>395</td>
</tr>
<tr>
<td>numWaitListed</td>
<td>20</td>
</tr>
<tr>
<td>totalStudents</td>
<td>410</td>
</tr>
</tbody>
</table>

• Remember: we do not re-declare numStudents
  – It is already known to be an int
  – The new value simply replaces the old value
1.1 Our first data type: \texttt{int}

- The computer executes your statements in sequence
  - One after the other
  - From the top, reading down

\begin{tabular}{|l|l|}
  \hline
  Variable Name & Value \\
  \hline
  numStudents & 395 \\
  numWaitListed & 20 \\
  totalStudents & 410 \\
  \hline
\end{tabular}

\begin{verbatim}
int numStudents = 390;
int numWaitListed = 20;
int totalStudents = numStudents + numWaitListed;
numStudents = numStudents + 5;
\end{verbatim}

- Other values are \textit{not} automatically updated
  - They have only been assigned their initial value
  - Code executes from the top, reading down
1.1 Our first data type: int

- The computer executes your statements in sequence
  - One after the other
  - From the top, reading down

```java
int numStudents = 390;
int numWaitListed = 20;
int totalStudents = numStudents + numWaitListed;

numStudents = numStudents + 5;

totalStudents = numStudents + numWaitListed;
```

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>numStudents</td>
<td>395</td>
</tr>
<tr>
<td>numWaitListed</td>
<td>20</td>
</tr>
<tr>
<td>totalStudents</td>
<td>410</td>
</tr>
</tbody>
</table>
1.1 Our first data type: int

- The computer executes your statements in sequence
  - One after the other
  - From the top, reading down

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>numStudents</td>
<td>395</td>
</tr>
<tr>
<td>numWaitListed</td>
<td>20</td>
</tr>
<tr>
<td>totalStudents</td>
<td>415</td>
</tr>
</tbody>
</table>

```c
int numStudents = 390;
int numWaitListed = 20;
int totalStudents = numStudents + numWaitListed;
numStudents = numStudents + 5;

totalStudents = numStudents + numWaitListed;
```
1.1b Exercise

• Video intro
  – Important to try things out in small doses
  – Story
    • Alice has 7 carrots
    • Bob has 3 more than Alice
    • Charles has 4 times as many as Alice
    • Also, Charles stole all of Bob's carrots
    • Diane has twice as many as Charles

• Question card: Translate the story to code
  – Print out how many Diane has at the end
    • System.out.println(dianeCarrots);

• Response: go over my solution
1.1c Now get input from the user

- Student asks question about how to make Alice's carrots a value input by the user
  - Much more on this later
  - But for now we use the `ArgsProcessor` stuff

- Show what happens when we try to use it
  - import the right thing with eclipse's help

- Meaningful prompting messages

- Run it a few times
1.2 Operations on int types

• Most operations work as expected
  – + plus
  – - minus
  – * times

• One operation usually surprises students and experienced programmers alike
  – Division using the / operator
  – Returns an int, so no fractional values are allowed
  – The value returned is as if the digits past the decimal point were dropped

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 / 2</td>
<td>2</td>
</tr>
</tbody>
</table>
1.2 Operations on int types

• Most operations work as expected
  – + plus
  – - minus
  – * times

• One operation usually surprises students and experienced programmers alike
  – Division using the / operator
  – Returns an int, so no fractional values are allowed
  – The value returned is as if the digits past the decimal point were dropped

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 / 2</td>
<td>2</td>
</tr>
<tr>
<td>3 / 2</td>
<td>1</td>
</tr>
</tbody>
</table>
1.2 Operations on int types

- Most operations work as expected
  - + plus
  - - minus
  - * times

- One operation usually surprises students and experienced programmers alike
  - Division using the / operator
  - Returns an int, so no fractional values are allowed
  - The value returned is as if the digits past the decimal point were dropped

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 / 2</td>
<td>2</td>
</tr>
<tr>
<td>3 / 2</td>
<td>1</td>
</tr>
<tr>
<td>5 / 2</td>
<td>2</td>
</tr>
</tbody>
</table>
1.2 Operations on int types

- Most operations work as expected
  - + plus
  - - minus
  - * times

- One operation usually surprises students and experienced programmers alike
  - Division using the / operator
  - Returns an int, so no fractional values are allowed
  - The value returned is as if the digits past the decimal point were dropped
  - We are not allowed to divide by 0

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 / 2</td>
<td>2</td>
</tr>
<tr>
<td>3 / 2</td>
<td>1</td>
</tr>
<tr>
<td>5 / 2</td>
<td>2</td>
</tr>
<tr>
<td>7 / 0</td>
<td>Error !</td>
</tr>
</tbody>
</table>
1.2 Operations on int types

- The `%` operator computes the remainder after division

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>x/y</th>
<th>x % y</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>
1.2 Operations on int types

- The % operator computes the remainder after division

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>x/y</th>
<th>x % y</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
1.2 Operations on int types

• The % operator computes the remainder after division

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>x/y</th>
<th>x % y</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

Why is this useful?
1.2 Operations on int types

• The % operator computes the remainder after division

• Examples:
  – Convert a large number of pennies into dollars and cents

    int pennies = 789321;

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>pennies</td>
<td>789321</td>
</tr>
</tbody>
</table>
1.2 Operations on int types

• The % operator computes the remainder after division

• Examples:
  – Convert a large number of pennies into dollars and cents

```c
int pennies = 789321;

int dollars = pennies / 100;
```

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>pennies</td>
<td>789321</td>
</tr>
<tr>
<td>dollars</td>
<td>7893</td>
</tr>
</tbody>
</table>
1.2 Operations on int types

• The % operator computes the remainder after division

• Examples:
  – Convert a large number of pennies into dollars and cents

```
int pennies = 789321;
int dollars = pennies / 100;
int cents = pennies % 100;
```

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>pennies</td>
<td>789321</td>
</tr>
<tr>
<td>dollars</td>
<td>7893</td>
</tr>
<tr>
<td>cents</td>
<td>21</td>
</tr>
</tbody>
</table>
1.3 Exercise

• Video intro
  – We see how to take a number of pennies and compute the dollars and cents contained therein
  – First we'll write code for that much and make sure it works.
  – Then we will incrementally improve the code
    • Allow quarters in change
    • Allow nickels and dimes too
1.3 Exercise

• **Question card**
  – **Write code that gives change as follows:**
    • Prompt the user for a number of pennies
      – Call this value `initialPennies`
    • Compute \( d \) as the number of dollars contained therein
      – Let’s assume the user trades in the appropriate number of pennies for the \( d \) dollars
    • Compute `penniesLeftAfterDollars` as the number of pennies that remain
    • Print out the dollars and cents that are contained in the `initialPennies`
  – **Try out your code and make sure that it works**
1.3 Exercise

• Question card
  – Now improve your code so that quarters can be given as well
    • Assume penniesLeftAfterDollars remain after dollars are given
    • How many quarters could be substituted for that many pennies? Call this value $q$ and print it out as well as the pennies left over after dollars and quarters are exchanged

• Example: 798 pennies
  – 7 dollars, leaving 98 penniesAfterCollars
  – $q=3$ quarters in 98 pennies
  – 23 pennies left over
1.3 Exercise

• Question card
  – Now improve your code so that change can be made
    • dollars
    • quarters
    • dimes
    • nickels
    • pennies left over
1.3 Exercise

• Example
  – 798 pennies initially
  – Should get
    • 7 dollars, leaving 98 cents
    • 3 quarters, leaving $98 - 75 = 98 \% 25 = 23$ cents
    • 2 dimes, leaving 3 cents
    • 0 nickels
    • 3 pennies
1.3 Exercise

• Video response
  – Goes over a correct solution line by line using the example from the setup
  – How would we verify that the answer we produced was correct?

• \(d \times 100 + q \times 25 + d \times 10 + n \times 5 + p\) should equal initialPennies
• Conversation about implementation of math on computers
  – If we have the largest integer value and add 1 to it, what happens?
    • Show this with screenflow movie
• We see computers of all sizes
  – Desktops, laptops, cell phones, smart watches
  – The basic hardware mechanisms of those computers can perform arithmetic differently
    • Largest and smallest values
• Java requires all computers to perform arithmetic Java's way
  – Because of Internet applets
  – A Java program runs the same way anywhere
  – Widespread portability and software reliability
1.4 Building Bigger Expressions

• We can combine smaller expressions into bigger
  – Using operators between smaller expressions
  – Using parentheses
    • To direct the order of evaluation
    • To improve the clarity of the expression

• My Dear Aunt Sally
  – Multiplication and Division have higher precedence
  – Addition and Subtraction have lower precedence
  – Evaluation is otherwise left-to-right
  – Parenthesized expressions have highest precedence
1.4 Building Bigger Expressions

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 + 4</td>
<td>7</td>
<td>As expected</td>
</tr>
</tbody>
</table>
## 1.4 Building Bigger Expressions

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>$3 + 4$</td>
<td>7</td>
<td>As expected</td>
</tr>
<tr>
<td>$3 \times 4 - 2$</td>
<td>10</td>
<td>$3 \times 4$ computed first</td>
</tr>
</tbody>
</table>
## 1.4 Building Bigger Expressions

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 + 4</td>
<td>7</td>
<td>As expected</td>
</tr>
<tr>
<td>3 * 4 − 2</td>
<td>10</td>
<td>3*4 computed first</td>
</tr>
<tr>
<td>3 + 4 * 2</td>
<td>11</td>
<td>4*2 computed first (MDAS)</td>
</tr>
</tbody>
</table>
### 1.4 Building Bigger Expressions

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>$3 + 4$</td>
<td>7</td>
<td>As expected</td>
</tr>
<tr>
<td>$3 * 4 - 2$</td>
<td>10</td>
<td>$3*4$ computed first</td>
</tr>
<tr>
<td>$3 + 4 * 2$</td>
<td>11</td>
<td>$4*2$ computed first (MDAS)</td>
</tr>
<tr>
<td>$3 + (4 * 2)$</td>
<td>11</td>
<td>Same result as above, but clearer. The parentheses are not necessary but they make the expression easier for humans to understand</td>
</tr>
</tbody>
</table>
## 1.4 Building Bigger Expressions

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>$3 + 4$</td>
<td>7</td>
<td>As expected</td>
</tr>
<tr>
<td>$3 \times 4 - 2$</td>
<td>10</td>
<td>$3 \times 4$ computed first</td>
</tr>
<tr>
<td>$3 + 4 \times 2$</td>
<td>11</td>
<td>$4 \times 2$ computed first (MDAS)</td>
</tr>
<tr>
<td>$3 + (4 \times 2)$</td>
<td>11</td>
<td>Same result as above, but clearer. The parentheses are not necessary but they make the expression easier for humans to understand</td>
</tr>
<tr>
<td>$(3 + 4) \times 2$</td>
<td>14</td>
<td>Parentheses affect the order</td>
</tr>
</tbody>
</table>
## 1.4 Building Bigger Expressions

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 + 4</td>
<td>7</td>
<td>As expected</td>
</tr>
<tr>
<td>3 * 4 – 2</td>
<td>10</td>
<td>3*4 computed first</td>
</tr>
<tr>
<td>3 + 4 * 2</td>
<td>11</td>
<td>4*2 computed first (MDAS)</td>
</tr>
<tr>
<td>3 + (4 * 2)</td>
<td>11</td>
<td>Same result as above, but clearer. The parentheses are not necessary but they make the expression easier for humans to understand</td>
</tr>
<tr>
<td>(3 + 4) * 2</td>
<td>14</td>
<td>Parentheses affect the order</td>
</tr>
<tr>
<td>4–3–2–1</td>
<td>–2</td>
<td>Should you use parentheses here to make this clearer?</td>
</tr>
</tbody>
</table>
# 1.4 Building Bigger Expressions

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>$3 + 4$</td>
<td>7</td>
<td>As expected</td>
</tr>
<tr>
<td>$3 \times 4 - 2$</td>
<td>10</td>
<td>$3 \times 4$ computed first</td>
</tr>
<tr>
<td>$3 + 4 \times 2$</td>
<td>11</td>
<td>$4 \times 2$ computed first (MDAS)</td>
</tr>
<tr>
<td>$3 + (4 \times 2)$</td>
<td>11</td>
<td>Same result as above, but clearer. The parentheses are not necessary but they make the expression easier for humans to understand</td>
</tr>
<tr>
<td>$(3 + 4) \times 2$</td>
<td>14</td>
<td>Parentheses affect the order</td>
</tr>
<tr>
<td>$4 - 3 - 2 - 1$</td>
<td>$-2$</td>
<td>Should you use parentheses here to make this clearer?</td>
</tr>
<tr>
<td>$4 - (3 - (2 - 1))$</td>
<td>2</td>
<td>To get this result, parentheses are necessary</td>
</tr>
</tbody>
</table>
1.4 Building Bigger Expressions

• To understand how an expression is computed, we draw its *evaluation tree*
• Evaluation proceeds from the leaves of the tree toward its root
• A node can be computed only when the values from all of its children are available

```java
int val = (2 + 3) / 2;
```

5
1.4 Building Bigger Expressions

- To understand how an expression is computed, we draw its *evaluation tree*
- Evaluation proceeds from the leaves of the tree toward its root
- A node can be computed only when the values from all of its children are available

```java
int val = (2 + 3) / 2;
```
1.4 Building Bigger Expressions

- To understand how an expression is computed, we draw its *evaluation tree*
- Evaluation proceeds from the leaves of the tree toward its root
- A node can be computed only when the values from all of its children are available
- Expression is complete, and value is stored

```c
int val = (2 + 3) / 2;
```
1.4 Building Bigger Expressions

\[4 - 3 - 2 - 1 \quad \text{vs.} \quad 4 - (3 - (2 - 1))\]
1.4 Building Bigger Expressions

• These large and complicated expressions are truly pedagogical examples
• We write programs not only for the computer
  – But for other humans to understand
• The computer has no problem with large, complicated expressions
  – But we humans do!
• Better to write simpler, easier-to-read expressions
  – If it takes you time to understand what you wrote, then what you write is too complicated
  – A good test is to show a friend your code and get critical feedback about its quality and readability
1.5 The double data type

- The int data type allows us to represent integer-valued items
  - How many students are in this class?
  - How many exams will be given?
  - How many words are in this sentence?

- We sometimes need value beyond the decimal point
  - The average age of students in this class?
  - The probability of “heads” on a coin flip

- We sometimes need really large or small values
  - \(6.0221413 \times 10^{23}\) (Avogadro’s number)
  - \(3 \times 10^{-9}\) seconds (Time for light to travel one meter)
1.5 The double data type

- The int data type allows us to represent integer-valued items
  - How many students are in this class?
  - How many exams will be given?
  - How many words are in this sentence?

- We sometimes need value beyond the decimal point
  - The average age of students in this class
  - The probability of “heads” on a coin flip
1.5 The `double` data type

- The `int` data type allows us to represent integer-valued items
  - How many students are in this class?
  - How many exams will be given?
  - How many words are in this sentence?

- We sometimes need value beyond the decimal point
  - The average age of students in this class
  - The probability of “heads” on a coin flip

- We sometimes need really large or small values
  - $6.0221413 \times 10^{23}$ (Avagadro’s number)
  - $3 \times 10^{-9}$ seconds (Time for light to travel one meter)
1.5 The `double` data type

- The syntax is similar
  - The type is different (`double`)
    - This is an unfortunate choice of name for a type
      - It refers to the amount of space used on a computer to represent the type.
    - We should think of it as a type that represents *rational* or *real* numbers

```c
double secsPerMeter = .000000003;
```
1.5 The `double` data type

- The syntax is similar
  - The type is different (`double`)
  - We can use scientific notation
    - The E character represents “times 10 to the”
    - You can also specify the E as lower-case e

```c
double secsPerMeter = 3E-9;
```
1.5 The `double` data type

- Most operations work as expected
  - `+` plus
  - `-` minus
  - `*` times

- As with `int`, division has some quirks for `double`
  - You can divide by 0
    - And get either Infinity or NaN (Not a Number)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.0 / 2.0</td>
<td>1.5</td>
</tr>
</tbody>
</table>
1.5 The `double` data type

- Most operations work as expected
  - + plus
  - - minus
  - * times

- As with `int`, division has some quirks for `double`
  - You can divide by 0
    - And get either Infinity or NaN (Not a Number)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.0 / 2.0</td>
<td>1.5</td>
</tr>
<tr>
<td>3 / 2</td>
<td>1</td>
</tr>
</tbody>
</table>
1.5 The double data type

- Most operations work as expected
  - + plus
  - - minus
  - * times

- As with int, division has some quirks for double
  - You can divide by 0
    - And get either Infinity or NaN (Not a Number)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.0 / 2.0</td>
<td>1.5</td>
</tr>
<tr>
<td>3 / 2</td>
<td>1</td>
</tr>
<tr>
<td>1.0 / 0.0</td>
<td>Infinity</td>
</tr>
</tbody>
</table>
1.5 The **double** data type

- Most operations work as expected
  - + plus
  - - minus
  - * times

- As with **int**, division has some quirks for **double**
  - You can divide by 0
    - And get either Infinity or NaN (Not a Number)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.0 / 2.0</td>
<td>1.5</td>
</tr>
<tr>
<td>3 / 2</td>
<td>1</td>
</tr>
<tr>
<td>1.0 / 0.0</td>
<td>Infinity</td>
</tr>
<tr>
<td>0.0 / 0.0</td>
<td>NaN</td>
</tr>
</tbody>
</table>
1.5 The double data type

- There are other surprises
  - The data type is inexact for some values
  - For nonrationals there is nothing we can do about this
    - How would you represent “pi” exactly?
  - For rationals, we can design a better data type
    - But the fastest arithmetic on the computer is approximate in terms of values, as shown below

- All of this is governed by an accepted standard
  - IEEE 754

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.5 + 0.25</td>
<td>0.75</td>
</tr>
<tr>
<td>0.1 + 0.2</td>
<td>0.30000000000000004</td>
</tr>
</tbody>
</table>
1.5 The double data type

- There are other surprises
  - The data type is inexact for some values
  - For nonrationals there is nothing we can do about this
    - How would you represent “pi” exactly?
  - For rationals, we can design a better data type
    - But the fastest arithmetic on the computer is approximate in terms of values, as shown below

- All of this is governed by an accepted standard
  - IEEE 754

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.5 + 0.25</td>
<td>0.75</td>
</tr>
<tr>
<td>0.1 + 0.2</td>
<td>0.30000000000000000004</td>
</tr>
</tbody>
</table>

It's a good idea to keep in mind the precision you want for a given computation, and round or truncate to reflect that precision. We will soon learn how to do this.
1.5 The double data type

• There are other surprises
  – The data type is inexact for some values
  – For nonrationals there is nothing we can do about this
    • How would you represent “pi” exactly?
  – For rationals, we can design a better data type
    • But the fastest arithmetic on the computer is approximate in terms of values, as shown below

• All of this is governed by an accepted standard
  – IEEE 754

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.5 + 0.25</td>
<td>0.75</td>
</tr>
<tr>
<td>0.1 + 0.2</td>
<td>0.3</td>
</tr>
</tbody>
</table>

It's a good idea to keep in mind the precision you want for a given computation, and round or truncate to reflect that precision. We will soon learn how to do this.
1.5 The double data type

- Languages provide math libraries with useful functions and values
  - Maximum and minimum
    - Math.max(1.3, 34)
    - Math.min(-5, 23)
1.5 The double data type

- Languages provide math libraries with useful functions and values
  - **Maximum and minimum**
    - `Math.max(1.3, 34)`
    - `Math.min(-5, 23)`
  - **Transcendental functions**
    - `sin, cos, tan, log, exp, sqrt`
    - Specified as `Math.sin(...), Math.cos(...), etc.`
  - **Random values**
    - `Math.random()`
      - Takes no input
      - Returns a pseudo-random number `r`, `0 ≤ r < 1`
  - **Rounding**
    - `Math.round(num)`
      - Rounds `num` to the nearest integer
      - Returns value of `long` type (like an `int`, but bigger)
  - **Constants**
    - `Math.PI` – closer to the actual value of $\pi$ than any other `double`
    - `Math.E` – closer to the actual value of $e$ than any other `double`
    - Many, many more. See this link
1.5 The double data type

- Languages provide math libraries with useful functions and values
  - **Maximum and minimum**
    - `Math.max(1.3, 34)`
    - `Math.min(-5, 23)`
  - **Transcendental functions**
    - `sin, cos, tan, log, exp, sqrt`
    - Specified as `Math.sin(...)`, `Math.cos(...)`, etc.
  - **Random values**: `Math.random()`
    - Takes no input
    - Returns a *pseudo*-random number $r$, $0 \leq r < 1$
1.5 The `double` data type

- Languages provide math libraries with useful functions and values
  - **Maximum and minimum**
    - `Math.max(1.3, 34)`
    - `Math.min(-5, 23)`
  - **Transcendental functions**
    - `sin`, `cos`, `tan`, `log`, `exp`, `sqrt`
    - Specified as `Math.sin(...)`, `Math.cos(...)`, etc.
  - **Random values:** `Math.random()`
    - Takes no input
    - Returns a pseudo-random number `r`, `0 ≤ r < 1`
  - **Rounding:** `Math.round(num)`
    - Rounds `num` to the nearest integer
    - Returns value of `long` type (like an `int`, but bigger)
1.5 The double data type

- Languages provide math libraries with useful functions and values
  - Maximum and minimum
    - `Math.max(1.3, 34)`
    - `Math.min(-5, 23)`
  - Transcendental functions
    - `sin`, `cos`, `tan`, `log`, `exp`, `sqrt`
    - Specified as `Math.sin(...)`, `Math.cos(...)`, etc.
  - Random values: `Math.random()`
    - Takes no input
    - Returns a pseudo-random number \( r, 0 \leq r < 1 \)
  - Rounding: `Math.round(num)`
    - Rounds `num` to the nearest integer
    - Returns value of `long` type (like an `int`, but bigger)
  - Constants
    - `Math.PI` – closer to the actual value of \( \pi \) than any other double
    - `Math.E` – closer to the actual value of \( e \) than any other double
- Many, many more. See [this link](#)
Randomness
- Run the random number generator and observe some of its outputs
- Compute the average of 10 such numbers

Pseudo-random
- What does this mean?
1.6 Mixed-type Expressions

- We can combine types in expressions where this makes sense
  - Here we look at int and double types
1.6 Mixed-type Expressions

- We can combine types in expressions where this makes sense
  - Here we look at int and double types
- If two types participate in an expression
  - The result is usually of the more general type

- For example, 3 becomes 3.0
- But the converse is not true
  - No way to represent 0.5 as an int
1.6 Mixed-type Expressions

• We can combine types in expressions where this makes sense
  – Here we look at int and double types

• If two types participate in an expression
  – The result is usually of the more general type

• The double type is more general than the int type
  – We can represent any int as a double
    • For example, 3 becomes 3.0
  – But the converse is not true
    • No way to represent 0.5 as an int
1.6 Mixed-type Expressions

<table>
<thead>
<tr>
<th>Expression</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>double radius = 2;</td>
<td>Allowed</td>
</tr>
</tbody>
</table>
# 1.6 Mixed-type Expressions

<table>
<thead>
<tr>
<th>Expression</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>double radius = 2;</code></td>
<td>Allowed</td>
</tr>
<tr>
<td><code>int area = 3.4;</code></td>
<td>Not allowed</td>
</tr>
</tbody>
</table>
1.6 Mixed-type Expressions

<table>
<thead>
<tr>
<th>Expression</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>double radius = 2;</td>
<td>Allowed</td>
</tr>
<tr>
<td>int area = 3.4;</td>
<td>Not allowed</td>
</tr>
<tr>
<td>int area = (int) 3.4;</td>
<td>Allowed, but we lose the data after the decimal point. The conversion requires the <em>explicit cast</em>, shown in the red box. <strong>Casts have very high precedence</strong></td>
</tr>
</tbody>
</table>
# 1.6 Mixed-type Expressions

<table>
<thead>
<tr>
<th>Expression</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>double radius = 2;</code></td>
<td>Allowed</td>
</tr>
<tr>
<td><code>int area = 3.4;</code></td>
<td>Not allowed</td>
</tr>
<tr>
<td><code>int area = (int) 3.4;</code></td>
<td>Allowed, but we lose the data after the decimal point. The conversion requires the <em>explicit cast</em>, shown in the red box. Casts have very high precedence</td>
</tr>
<tr>
<td><code>int avg = (2 + 3) / 2;</code></td>
<td>Allowed, but the result is <em>not</em> 2.5! The result is computed in integer arithmetic, so the value assigned to <code>avg</code> is 2</td>
</tr>
</tbody>
</table>
### 1.6 Mixed-type Expressions

<table>
<thead>
<tr>
<th>Expression</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>double radius = 2;</td>
<td>Allowed</td>
</tr>
<tr>
<td>int area = 3.4;</td>
<td>Not allowed</td>
</tr>
<tr>
<td>int area = (int) 3.4;</td>
<td>Allowed, but we lose the data after the decimal point. The conversion requires the <em>explicit cast</em>, shown in the red box. Casts have very high precedence</td>
</tr>
<tr>
<td>int avg = (2 + 3) / 2;</td>
<td>Allowed, but the result is <em>not</em> 2.5! The result is computed in integer arithmetic, so the value assigned to <code>avg</code> is 2</td>
</tr>
<tr>
<td>double avg = (2 + 3) / 2;</td>
<td>The right-hand side is computed first, resulting in 2, and that value is then stored in the variable, which results in the value 2.0 for <code>avg</code></td>
</tr>
</tbody>
</table>
## 1.6 Mixed-type Expressions

<table>
<thead>
<tr>
<th>Expression</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>double radius = 2;</td>
<td>Allowed</td>
</tr>
<tr>
<td>int area = 3.4;</td>
<td>Not allowed</td>
</tr>
<tr>
<td>int area = (int) 3.4;</td>
<td>Allowed, but we lose the data after the decimal point. The conversion requires the <em>explicit cast</em>, shown in the red box. Casts have very high precedence</td>
</tr>
<tr>
<td>int avg = (2 + 3) / 2;</td>
<td>Allowed, but the result is <em>not</em> 2.5! The result is computed in integer arithmetic, so the value assigned to avg is 2</td>
</tr>
<tr>
<td>double avg = (2 + 3) / 2;</td>
<td>The right-hand side is computed first, resulting in 2, and that value is then stored in the variable, which results in the value 2.0 for avg</td>
</tr>
</tbody>
</table>
1.6 Mixed-type Expressions

- To understand how an expression is computed, we can draw its evaluation tree

```java
double avg = (double) (2 + 3) / 2;
```

5
1.6 Mixed-type Expressions

- To understand how an expression is computed, we can draw its \textit{evaluation tree}

\[
double \ avg = (double) (2 + 3) / 2;
\]
1.6 Mixed-type Expressions

- To understand how an expression is computed, we can draw its evaluation tree

```java
double avg = (double) (2 + 3) / 2;
```

```
            /
           /  
          /    
         /     
        /      
       /       
      /        
     /         
    2.5       5
   /  
  5.0
 /  
2.5
```
1.6 Mixed-type Expressions

- To understand how an expression is computed, we can draw its *evaluation tree*

```java
double avg = (double) (2 + 3) / 2;
```

```
 5
  /
 5.0
  |
 2.5
```
1.6 Mixed-type Expressions

• What value results from the following expression?

\[(\text{double}) \frac{1}{2} + \frac{2}{3} + \frac{3}{4}\]
1.6 Mixed-type Expressions

• What value results from the following expression?

\[(\text{double}) \frac{1}{2} + \frac{2}{3} + \frac{3}{4}\]

1.0
1.6 Mixed-type Expressions

- What value results from the following expression?

\[(\text{double}) \frac{1}{2} + \frac{2}{3} + \frac{3}{4}\]

\[1.0 + 0.5 = 1.5\]
1.6 Mixed-type Expressions

• What value results from the following expression?

\[(\text{double}) \frac{1}{2} + \frac{2}{3} + \frac{3}{4}\]
1.6 Mixed-type Expressions

- What value results from the following expression?

\[(\text{double})\ 1 \div 2\ +\ 2 \div 3\ +\ 3 \div 4\]
1.6 Mixed-type Expressions

• What value results from the following expression?

$$\text{(double)} \frac{1}{2} + \frac{2}{3} + \frac{3}{4}$$

1.0

0.5

0

0
1.6 Mixed-type Expressions

• What value results from the following expression?

```
(double) 1 / 2 + 2 / 3 + 3 / 4
```

1.0

0.5

0.5

0.5

0.5
1.7 Strings

- A useful data type for representing text
- String *literals* are specified using double quotes
  - "Hello"
  - "This is a string."
  - " " (an empty string)
1.7 Strings

- A useful data type for representing text
- String *literals* are specified using double quotes
  - "Hello"
  - "This is a string."
  - " " (an empty string)
- The only keyboard operator is +
  - The symbol means *concatenation*
  - It is overloaded from arithmetic for int and double types
1.7 Strings

- A useful data type for representing text
- String *literals* are specified using double quotes
  - "Hello"
  - "This is a string."
  - "" (an empty string)
- The only keyboard operator is +
  - The symbol means *concatenation*
  - It is overloaded from arithmetic for int and double types

<table>
<thead>
<tr>
<th>Example</th>
<th>Result, if printed</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Hello&quot; + &quot;there&quot;</td>
<td>Hello there</td>
</tr>
<tr>
<td>&quot;Hello &quot; + &quot;there&quot;</td>
<td>Hello there</td>
</tr>
<tr>
<td>&quot; &quot; + &quot; &quot; + &quot; &quot;</td>
<td>Hello there</td>
</tr>
</tbody>
</table>
1.7 Strings

• Declarations and assignments are the same as before.

```java
String s1 = "ei";
```

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1</td>
<td>ei</td>
</tr>
</tbody>
</table>
1.7 Strings

• Declarations and assignments are the same as before. Use of the capital S is important!

String s1 = "ei";

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1</td>
<td>ei</td>
</tr>
</tbody>
</table>
1.7 Strings

• Declarations and assignments are the same as before. Use of the capital S is important!

```java
String s1 = "ei";
String s2 = s1 + s1;
```

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1</td>
<td>ei</td>
</tr>
<tr>
<td>s2</td>
<td>eieie</td>
</tr>
</tbody>
</table>

1.7 Strings

• Declarations and assignments are the same as before. Use of the capital S is important!

```
String s1 = "ei";
String s2 = s1 + s1;
String s3 = s2 + "o";
```

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1</td>
<td>ei</td>
</tr>
<tr>
<td>s2</td>
<td>eiei</td>
</tr>
<tr>
<td>s3</td>
<td>eieio</td>
</tr>
</tbody>
</table>
### 1.7 Strings

- Conversion from other expression types to String produces the text form of their values.
- Given a String $s$, $s.length()$ returns the length of the string in characters.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>String size = 5;</td>
<td>Not allowed</td>
</tr>
</tbody>
</table>
1.7 Strings

- Conversion from other expression types to String produces the text form of their values.
- Given a String $s$, $s.length()$ returns the length of the string in characters.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>String size = 5;</td>
<td>Not allowed</td>
</tr>
<tr>
<td>String size = &quot;&quot; + 5;</td>
<td>Allowed, result is &quot;5&quot;</td>
</tr>
</tbody>
</table>
1.7 Strings

• Conversion from other expression types to String produces the text form of their values.
• Given a String $s$, $s.length()$ returns the length of the string in characters.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>String size = 5;</td>
<td>Not allowed</td>
</tr>
<tr>
<td>String size = &quot;&quot; + 5;</td>
<td>Allowed, result is &quot;5&quot;</td>
</tr>
<tr>
<td>int inches = 54; double feet = inches / 12.0; String msg = feet + &quot; ft&quot;;</td>
<td>Measurement in inches Conversion to feet &quot;4.5 ft&quot; is the result</td>
</tr>
</tbody>
</table>
1.7 Strings

- Conversion from other expression types to String produces the text form of their values.
- Given a String $s$, $s.length()$ returns the length of the string in characters.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>String size = 5;</td>
<td>Not allowed</td>
</tr>
<tr>
<td>String size = &quot;&quot; + 5;</td>
<td>Allowed, result is &quot;5&quot;</td>
</tr>
<tr>
<td>int inches = 54; double feet = inches / 12.0; String msg = feet + &quot; ft&quot;;</td>
<td>Measurement in inches Conversion to feet &quot;4.5 ft&quot; is the result</td>
</tr>
<tr>
<td>int length = msg.length()</td>
<td>Returns the number of characters in the string. In this case, 6 is returned.</td>
</tr>
</tbody>
</table>
1.8 The boolean data type

- Useful for representing two-valued concepts

- Named after George Bool – 19th century logician and mathematician
  - But lower case b is important

- Only two constants (lower case is important)
  - true
  - false
1.8 The boolean data type

- Useful for representing two-valued concepts
  - True vs. false
  - Heads vs. tails
  - Open vs. closed
1.8 The boolean data type

- Useful for representing two-valued concepts
  - True vs. false
  - Heads vs. tails
  - Open vs. closed

- Named after George Bool
  - 19th century logician and mathematician
  - But lower case b is important
1.8 The boolean data type

- Useful for representing two-valued concepts
  - True vs. false
  - Heads vs. tails
  - Open vs. closed

- Named after George Bool
  - 19th century logician and mathematician
  - But lower case b is important

- Only two constants (lower case is important)
  - true
  - false
1.8 The boolean data type

- Mathematical operators work as expected, though some are spelled strangely
  - `==` for equality
  - `!=` for inequality
  - `<=` for ≤, etc.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 &gt; 4</td>
<td>false</td>
</tr>
</tbody>
</table>
1.8 The boolean data type

- Mathematical operators work as expected, though some are spelled strangely
  - `==` for equality
  - `!=` for inequality
  - `<=` for ≤, etc.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 &gt; 4</td>
<td>false</td>
</tr>
<tr>
<td>3.5 &lt; 7</td>
<td>true</td>
</tr>
</tbody>
</table>
1.8 The boolean data type

- Mathematical operators work as expected, though some are spelled strangely
  - == for equality
  - != for inequality
  - <= for $\leq$, etc.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 &gt; 4</td>
<td>false</td>
</tr>
<tr>
<td>3.5 &lt; 7</td>
<td>true</td>
</tr>
</tbody>
</table>

The usual rules of conversion apply here. The int 7 is converted to the double 7.0 and the comparison with 3.5 is then made.
1.8 The boolean data type

- Mathematical operators work as expected, though some are spelled strangely
  - `==` for equality
  - `!=` for inequality
  - `<=` for $\leq$, etc.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$3 &gt; 4$</td>
<td>false</td>
</tr>
<tr>
<td>$3.5 &lt; 7$</td>
<td>true</td>
</tr>
<tr>
<td>$3 == 3$</td>
<td>true</td>
</tr>
</tbody>
</table>
• Mathematical operators work as expected, though some are spelled strangely
  – `==` for equality
  – `!=` for inequality
  – `<=` for \( \leq \), etc.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 &gt; 4</td>
<td>false</td>
</tr>
<tr>
<td>3.5 &lt; 7</td>
<td>true</td>
</tr>
<tr>
<td>3 == 3</td>
<td>true</td>
</tr>
<tr>
<td>3 != 3</td>
<td>false</td>
</tr>
</tbody>
</table>
### 1.8 The boolean data type

- Mathematical operators work as expected, though some are spelled strangely
  - `==` for equality
  - `!=` for inequality
  - `<=` for ≤, etc.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 &gt; 4</td>
<td>false</td>
</tr>
<tr>
<td>3.5 &lt; 7</td>
<td>true</td>
</tr>
<tr>
<td>3 == 3</td>
<td>true</td>
</tr>
<tr>
<td>3 != 3</td>
<td>false</td>
</tr>
<tr>
<td>2 &lt;= 1 + 1</td>
<td>true</td>
</tr>
</tbody>
</table>
1.8 The boolean data type

- Mathematical operators work as expected, though some are spelled strangely
  – == for equality
  – != for inequality
  – <= for ≤, etc.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 &gt; 4</td>
<td>false</td>
</tr>
<tr>
<td>3.5 &lt; 7</td>
<td>true</td>
</tr>
<tr>
<td>3 == 3</td>
<td>true</td>
</tr>
<tr>
<td>3 != 3</td>
<td>false</td>
</tr>
<tr>
<td>2 &lt;= 1 + 1</td>
<td>true</td>
</tr>
<tr>
<td>true == true</td>
<td>true</td>
</tr>
</tbody>
</table>
1.8 The boolean data type

- Mathematical operators work as expected, though some are spelled strangely
  - == for equality
  - != for inequality
  - <= for ≤, etc.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 &gt; 4</td>
<td>false</td>
</tr>
<tr>
<td>3.5 &lt; 7</td>
<td>true</td>
</tr>
<tr>
<td>3 == 3</td>
<td>true</td>
</tr>
<tr>
<td>3 != 3</td>
<td>false</td>
</tr>
<tr>
<td>2 &lt;= 1 + 1</td>
<td>true</td>
</tr>
<tr>
<td>true == true</td>
<td>true</td>
</tr>
<tr>
<td>true == false</td>
<td>false</td>
</tr>
</tbody>
</table>
1.8 The boolean data type

- Logical operators are also available.
  - **Complement**
    - \(!p\) is true if \(p\) is false
    - \(!p\) is false if \(p\) is true

  - **And**
    - \(p && q\) is true if both \(p\) and \(q\) have the value true
  - **Or**
    - \(p || q\) is true if either \(p\) or \(q\) is true, or if both \(p\) and \(q\) are true
    - \(p |\| q\) is false only if \(p\) is false and \(q\) is false

  - This is different from the way we use or in English
  - I am going to the store or I am going to a movie
    - Means one or the other, but not both
1.8 The boolean data type

• Logical operators are also available.
  – **Complement**
    • \( !p \) is true if \( p \) is false
    • \( !p \) is false if \( p \) is true
  – **And**
    • \( p \land q \) is true if both \( p \) and \( q \) have the value true
    • Works as expected for English meaning of *and*
1.8 The boolean data type

- Logical operators are also available.
  - **Complement**
    - \( \neg p \) is true if \( p \) is false
    - \( \neg p \) is false if \( p \) is true
  - **And**
    - \( p \land q \) is true if both \( p \) and \( q \) have the value true
    - Works as expected for English meaning of *and*
  - **Or**
    - \( p \lor q \) is true if either \( p \) or \( q \) is true, or if both \( p \) and \( q \) are true
    - \( p \lor q \) is false only if \( p \) is false and \( q \) is false
1.8 The boolean data type

- Logical operators are also available.
  - **Complement**
    - \( \neg p \) is true if \( p \) is false
    - \( \neg p \) is false if \( p \) is true
  - **And**
    - \( p \land q \) is true if both \( p \) and \( q \) have the value true
    - Works as expected for English meaning of *and*
  - **Or**
    - \( p \lor q \) is true if either \( p \) or \( q \) is true, or if both \( p \) and \( q \) are true
    - \( p \lor q \) is false only if \( p \) is false and \( q \) is false
    - This is different from the way we use *or* in English
      - I am going to the store or I am going to a movie
      - Means one or the other, but not both
1.8 The boolean data type

- Examples

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>p</td>
<td>!p</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>p</td>
<td>!p</td>
<td>true</td>
</tr>
</tbody>
</table>
1.8 The boolean data type

- **Examples**

<table>
<thead>
<tr>
<th>p</th>
<th>q</th>
<th>Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td></td>
<td>!p</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td></td>
<td>!p</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>p</td>
<td></td>
</tr>
</tbody>
</table>
1.8 The boolean data type

- **Examples**

<table>
<thead>
<tr>
<th>p</th>
<th>q</th>
<th>Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td></td>
<td>!p</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td></td>
<td>!p</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>p</td>
<td></td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>p &amp;&amp; q</td>
<td>false</td>
</tr>
</tbody>
</table>
1.8 The boolean data type

• Examples

<table>
<thead>
<tr>
<th>p</th>
<th>q</th>
<th>Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td></td>
<td>!p</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td></td>
<td>!p</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>p</td>
<td></td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>p &amp;&amp; q</td>
<td>false</td>
</tr>
<tr>
<td></td>
<td></td>
<td>p</td>
<td></td>
</tr>
</tbody>
</table>
1.8 The boolean data type

- **Examples**

<table>
<thead>
<tr>
<th>p</th>
<th>q</th>
<th>Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td></td>
<td>!p</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td></td>
<td>!p</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>p</td>
<td></td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>p &amp;&amp; q</td>
<td>false</td>
</tr>
<tr>
<td></td>
<td>p</td>
<td></td>
<td>!p</td>
</tr>
<tr>
<td></td>
<td>p &amp;&amp; !p</td>
<td>false</td>
<td></td>
</tr>
</tbody>
</table>
1.8 The boolean data type

- Examples

<table>
<thead>
<tr>
<th>p</th>
<th>q</th>
<th>Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td></td>
<td>!p</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td></td>
<td>!p</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>p</td>
<td></td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>p &amp;&amp; q</td>
<td>false</td>
</tr>
<tr>
<td></td>
<td></td>
<td>p</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>p &amp;&amp; !p</td>
<td>false</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>p</td>
<td></td>
</tr>
</tbody>
</table>
## 1.8 The boolean data type

### Examples

<table>
<thead>
<tr>
<th>p</th>
<th>q</th>
<th>Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td></td>
<td>![p]</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>!p</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>p</td>
<td></td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>p &amp;&amp; q</td>
<td>false</td>
</tr>
<tr>
<td></td>
<td>p</td>
<td></td>
<td>!p</td>
</tr>
<tr>
<td></td>
<td>p &amp;&amp; !p</td>
<td>false</td>
<td></td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>p</td>
<td></td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>(p</td>
<td></td>
</tr>
</tbody>
</table>
1.9 Conclusion

- **What is a type?**
  - A classification of items in the world based on
    - The kinds of things an item can do
    - The
    - integers, real numbers, character strings, true/false values
- **Types are also defined by the operations associated with them**
- **Data types in computer science**
  - Represent values of interest
  - Allow operations that make sense
- **Built in data types (in this module)**
  - For representing integers, interest rates, names, addresses, facts
- **Later, we design our own types**
  - Bank account, hockey player, song
- **The best choice of data type is not always obvious**
  - You may start with one and change your mind later
- **There are rarely right or wrong choices**
  - But a choice will have its advantages and disadvantages
  - Picking a suitable data type is an engineering process and is based on currently understood constraints and best planning for the future. The future can at times seem uncertain.