Integer Types

- Different types allow different ranges of values
  - `byte` – 8 bit number: -128 to 127
  - `short` – 16 bit number: -32768 to 32767
  - `int` – 32 bit number ≈ -2,000,000,000 to 2,000,000,000
  - `long` – 64 bit number ≈ -8 quintillion to 8 quintillion
    8,000,000,000,000,000,000

- Every bit doubles the range of numbers
Type Conversions

- It is acceptable to assign a value from a smaller type to a variable of a larger type
  - byte to short
  - short to int
  - int to long

- But other way is not OK without a "type cast"

```java
int x;
long y = 6L;
x = (int) y; //OK
```

No data loss will occur.
Termed: “widening”

Data loss may occur.
Termed: “shortening”
Type Cast

- Forced type conversion
- Syntax: `(type) expression`
- Examples:
  - If `y` is a `long` variable, `(int) y` is an `int` value
  - If `b` is a `short` variable, `(byte) b` is a `byte` value

Although allowed by the Java language data loss may occur due to “shortening”.
Real Number Types

- Different types allow different ranges
  - float: 7 digits of precision
  - double: 15 digits of precision

- Caution: not all numbers in range can be represented by floating point numbers
  - float and int use same storage
  - But 1,234,567,089 is an int but not a float!

WHY???
Type Conversions

- Integer values can be assigned to floating point variables – remember caution!
  - Some less significant digits may be lost due to limited precision.
- Floating point values can be assigned to integer variables with a cast
  - Truncation of the fractional, (decimal) quantity will occur.
Literals

- **byte**: number literal in valid range
- **short**: number literal in valid range
- **int**: number literal in valid range
- **long**: 32L or 32l
- **float**: 18f, 18.f, 18.0f, 1.8e1f, .18E2f
- **double**: 18, 18., 18.0, 1.8e1, .18E2

also use code `d` or `D` for `double`, `f` or `F` for `float`
To avoid “magic numbers” in code that reader might not understand
   Example: literal constants in tax computations

- Literals will not change during execution.
- Use constant identifiers to declare these values

```java
static final float SALES_TAX = 0.07f;
```

- Keyword `final` indicates that the value cannot change
- Use `static` so that all objects have access
Methods for Primitive Data

- There are several classes that provide methods (static) that operate on primitive data.
- Example: class **Math** provides general mathematical utilities
  ```java
  int diffXY = Math.abs(x - y);
  ```
- Other **wrapper classes** can hold values (for purposes needed later) and also provide functions.
- Wrapper classes: **Byte, Short, Integer, Long, Float** and **Double**.
Use `Integer` method `parseInt` to convert String to int

- `String s;`
- `int numCalls;`
- `s = kb.readLine(); //read line from input`
- `numCalls = Integer.parseInt(s); //looks for int in s`

Parsing means to separate a string into its parts

Method requires an integer
- String must be a number
- If String contains anything but a valid integer an exception will be thrown.
Steps

- Read string
- Create `Double` object from string
  ```java
  Double d = Double.valueOf(s);
  ```
- Extract double value from `Double` object
  ```java
  double x = d.doubleValue();
  ```

Combine into one statement:

```java
double x = Double.valueOf(s).doubleValue();
```
Simple Output Alignment

- Print spaces after strings to align next output under a heading label
  - Define a String constant for spacing:
    ```java
    final String SPACES = "                                       ";
    ```
  - Given the following sample heading labels:
    ```java
    Label1    Label2
    ```
  - The first output field, (under `Label1`) has 10 columns for output.
  - If the value printed only takes up 6 columns then 4 spaces must be printed to align
    the next output under the next heading label, `Label2`.
  - The code below correctly pads after the value output for alignment:
    ```java
    myOutput.print(value1);
    myOutput.print(SPACES.substring(0, 10 - value1.length()));
    ```
  - If `value1` is numeric it must be converted to a String:
    ```java
    myOutput.print(value1);
    myOutput.print(SPACES.substring(0, 10 - String.valueOf(value1).length()));
    ```
The DecimalFormat class provides output format control for doubles.

- Instantiate a DecimalFormat object passing a format string to the constructor:
  ```java
  DecimalFormat df = new DecimalFormat("#0.00");
  ```
- The ‘#’ symbol represents space for a digit, but the digit is not output if it is a leading or trailing zero.
- The ‘0’ symbol represents a digit. A zero is output even if it is a leading or trailing zero.
- Other symbols, (e.g., ‘$’) may be included in the format string. Commas may be included to indicate digit grouping separations.
- Example usage:
  ```java
  DecimalFormat df = new DecimalFormat("#0.00");
  Double d = 1.1;
  System.out.println(df.format(d));
  ```
- Outputs:
  ```java
  1.10
  ```
  - The underscore represents the leading space output by the ‘#’ placeholder.