Unit 2, Part 1

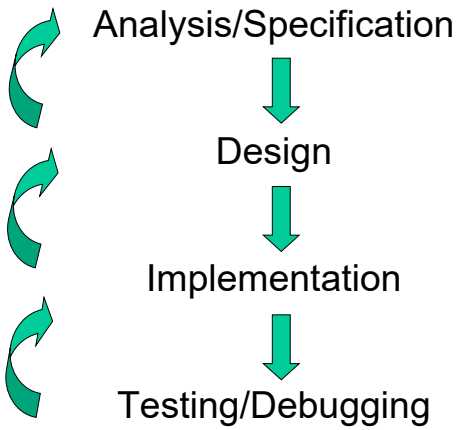
Primitive Data, Variables, and Expressions;

Simple Conditional Execution

Computer Science S-111
Harvard University

David G. Sullivan, Ph.D.

Overview of the Programming Process



Example Problem: Adding Up Your Change

- Let's say that we have a bunch of coins of various types, and we want to figure out how much money we have.
- Let's begin the process of developing a program that does this.

Step 1: Analysis and Specification

- *Analyze* the problem (making sure that you understand it), and *specify* the problem requirements clearly and unambiguously.
- Describe exactly *what* the program will do, without worrying about *how* it will do it.

Step 2: Design

- Determine the necessary algorithms (and possibly other aspects of the program) and sketch out a design for them.
- This is where we figure out *how* the program will solve the problem.
- Algorithms are often designed using *pseudocode.*
  - more informal than an actual programming language
  - allows us to avoid worrying about the *syntax* of the language
  - example for our change-adder problem:

```
get the number of quarters
get the number of dimes
get the number of nickels
get the number of pennies
compute the total value of the coins
output the total value
```

Step 3: Implementation

- Translate your design into the programming language.

  pseudocode → code

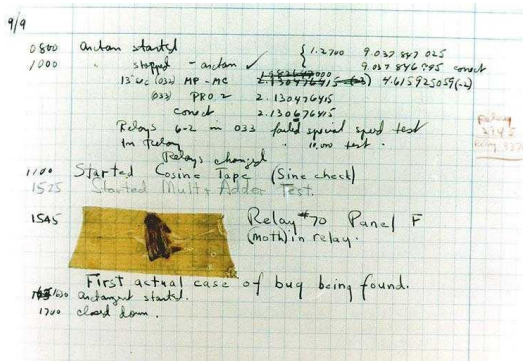
- We need to learn more Java before we can do this!
- Here's a portion or *fragment* of a Java program for computing the value of a particular collection of coins:

```
quarters = 10;
dimes = 3;
nickels = 7;
pennies = 6;

cents = 25*quarters + 10*dimes + 5*nickels + pennies;
System.out.println("Your total in cents is:");
System.out.println(cents);
```

- In a moment, we'll use this fragment to examine some of the fundamental building blocks of a Java program.

Step 4: Testing and Debugging

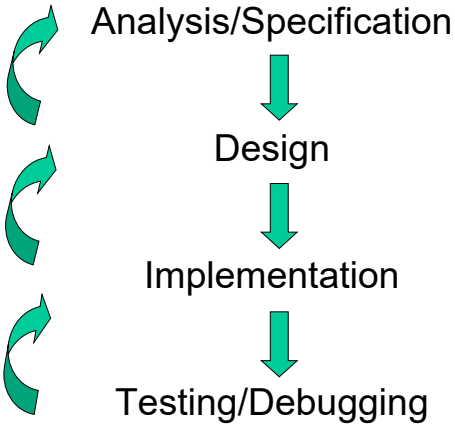
- A *bug* is an error in your program.
- *Debugging* involves finding and fixing the bugs.



The first program bug! Found by Grace Murray Hopper at Harvard.
(http://www.hopper.navy.mil/grace/grace.htm)

- Testing – trying the programs on a variety of inputs – helps us to find the bugs.

Overview of the Programming Process



Program Building Blocks: Literals

```
quarters = 10;
dimes = 3;
nickels = 7;
pennies = 6;

cents = 25*quarters + 10*dimes + 5*nickels + pennies;
System.out.println("Your total in cents is:");
System.out.println(cents);
```

- *Literals* specify a particular value.
- They include:
  - string literals: `"Your total in cents is:"`
    - are surrounded by double quotes
  - numeric literals: `25` `3.1416`
    - commas are not allowed!

Program Building Blocks: Variables

```
quarters = 10;
dimes = 3;
nickels = 7;
pennies = 6;

cents = 25*quarters + 10*dimes + 5*nickels + pennies;
System.out.println("Your total in cents is:");
System.out.println(cents);
```

- We've already seen that variables are named memory locations that are used to store a value:

- Variable names must follow the rules for *identifiers* (see previous notes).

Program Building Blocks: Statements

```
quarters = 10;
dimes = 3;
nickels = 7;
pennies = 6;

cents = 25*quarters + 10*dimes + 5*nickels + pennies;
System.out.println("Your total in cents is:");
System.out.println(cents);
```

- In Java, a single-line statement typically ends with a semi-colon.
- Later, we will see examples of statements that contain other statements!

Program Building Blocks: Expressions

```
quarters = 10;
dimes = 3;
nickels = 7;
pennies = 6;

cents = 25*quarters + 10*dimes + 5*nickels + pennies;
System.out.println("Your total in cents is:");
System.out.println(cents);
```

- *Expressions* are pieces of code that evaluate to a value.
- They include:
  - literals, which evaluate to themselves
  - variables, which evaluate to the value that they represent
  - combinations of literals, variables, and *operators*:

    `25*quarters + 10*dimes + 5*nickels + pennies`

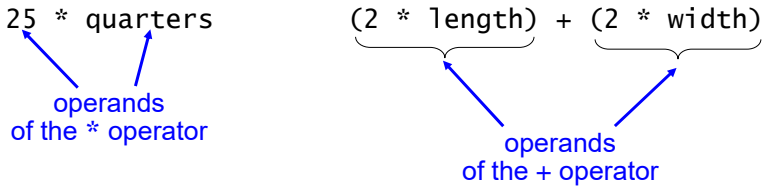
Program Building Blocks: Expressions (cont.)

- Numerical operators include:

`+`	addition
`-`	subtraction
`*`	multiplication
`/`	division
`%`	modulus or mod: gives the remainder of a division

example: `11 % 3` evaluates to `2`

- Operators are applied to *operands:*



Evaluating Expressions

- With expressions that involve more than one mathematical operator, the usual order of operations applies.
  - example:

```
  3 + 4 * 3 / 2 - 7
=
=
=
=
```

- Use parentheses to:
  - force a different order of evaluation
    - example:

```
radius = circumference / (2 * pi);
```

  - make the standard order of operations obvious!

Evaluating Expressions with Variables

- When an expression includes variables, they are first replaced with their current value.
- Example: recall our code fragment:

```
quarters = 10;
dimes = 3;
nickels = 7;
pennies = 6;
```

```
cents = 25*quarters + 10*dimes + 5*nickels + pennies;
      = 25*   10    + 10*  3   + 5*   7    + 6
      =    250      + 10*  3   + 5*   7    + 6
      =    250      + 30       + 5*   7    + 6
      =    250      + 30       + 35        + 6
      =         280             + 35        + 6
      =                 315                 + 6
      =                          321
```

`println` Statements Revisited

- Recall our earlier syntax for `println` statements:

```
System.out.println("text");
```

- Here is a more complete version:

```
System.out.println(expression);
```

any type of expression, not just text

- Examples:

```
System.out.println(3.1416);
System.out.println(2 + 10 / 5);
System.out.println(cents);     // a variable
System.out.println("cents");   // a string
```

println Statements Revisited (cont.)

- The expression is first evaluated, and then the value is printed.

```
System.out.println(2 + 10 / 5);
```

```
System.out.println(4);          // output: 4
```

```
System.out.println(cents);
```

```
System.out.println(321);        // output: 321
```

```
System.out.println("cents");
```

```
System.out.println("cents");    // output: cents
```

- Note that the surrounding quotes are *not* displayed when a string is printed.

println Statements Revisited (cont.)

- Another example:

```
System.out.println(10*dimes + 5*nickels);
```

```
System.out.println(10*3 + 5*7);
```

```
System.out.println(65);
```

Data Types

- A *data type* is a set of related data values.
  - examples:
    - integers
    - strings
    - characters
- Every data type in Java has a name that we can use to identify it.

Commonly Used Data Types for Numbers

- `int`
  - used for integers
  - examples: `25` `-2`
- `double`
  - used for real numbers (ones with a fractional part)
  - examples: `3.1416` `-15.2`
  - used for *any* numeric literal with a decimal point, even if it's an integer:

    `5.0`
  - also used for *any* numeric literal written in scientific notation

    `3e8` `-1.60e-19`

    more generally:

    $n \times 10^p$ is written `nep`

Incorrect Change-Adder Program

```
/*
 * ChangeAdder.java
 * Dave Sullivan (dgs@cs.bu.edu)
 * This program determines the value of some coins.
 */

public class ChangeAdder {
    public static void main(String[] args) {
        quarters = 10;
        dimes = 3;
        nickels = 7;
        pennies = 6;

        // compute and print the total value
        cents = 25*quarters + 10*dimes + 5*nickels + pennies;
        System.out.print("total in cents is: ");
        System.out.println(cents);
    }
}
```

Declaring a Variable

- Java requires that we specify the *type* of a variable before attempting to use it.
- This is called *declaring* the variable.
  - syntax:

    type name;
  - examples:

```
int count;          // will hold an integer
double area;        // will hold a real number
```

- A variable declaration can also include more than one variable of the same type:

```
int quarters, dimes;
```

Assignment Statements

- Used to give a value to a variable.
- Syntax:

  variable = *expression*;

  = is known as the *assignment operator.*
- Examples:

```
int quarters = 10;    // declaration plus assignment

// declaration first, assignment later
int cents;
cents = 25*quarters + 10*dimes + 5*nickels + pennies;

// can also use to change the value of a variable
quarters = 15;
```

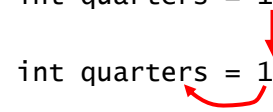
Corrected Change-Adder Program

```
/*
 * ChangeAdder.java
 * Dave Sullivan (dgs@cs.bu.edu)
 * This program determines the value of some coins.
 */

public class ChangeAdder {
    public static void main(String[] args) {
        int quarters = 10;
        int dimes = 3;
        int nickels = 7;
        int pennies = 6;
        int cents;

        // compute and print the total value
        cents = 25*quarters + 10*dimes + 5*nickels + pennies;
        System.out.print("total in cents is: ");
        System.out.println(cents);
    }
}
```

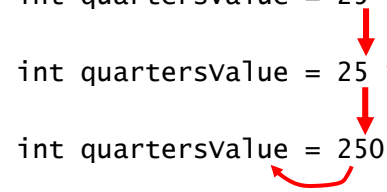
Assignment Statements (cont.)

- Steps in executing an assignment statement:
  1) evaluate the expression on the right-hand side of the =
  2) assign the resulting value to the variable on the left-hand side of the =
- Examples:

```
int quarters = 10;

int quarters = 10;    // 10 evaluates to itself!
```



```
int quartersValue = 25 * quarters;

int quartersValue = 25 * 10;

int quartersValue = 250;
```



Assignment Statements (cont.)

- An assignment statement does not create a permanent relationship between variables.
- Example: consider the following code fragment

```
int x = 10;
int y = x + 2;
System.out.println(y);
x = 20;
System.out.println(y);
```

it outputs:

  - changing the value of x does *not* change the value of y!
- You can only change the value of a variable by assigning it a new value.

Assignment Statements (cont.)

- As the values of variables change, it can be helpful to picture what's happening in memory.
- Examples:

```
int num1;
int num2 = 120;
```

undefined

num1 ? num2 120

```
num1 = 50;
```

after the assignment at left, we get:

num1 50 num2 120

```
num1 = num2 * 2;
       120  * 2
         240
```

num1 240 num2 120

```
num2 = 60;
```

num1 240 num2 60

The value of num1 is unchanged!

Assignment Statements (cont.)

- A variable can appear on both sides of the assignment operator!
- Example (fill in the missing values):

```
int sum = 13;
int val = 30;
```

sum 13 val 30

```
sum = sum + val;
```

sum val

```
val = val * 2;
```

sum val

Operators and Data Types

- Each data type has its own set of operators.
  - the `int` version of an operator produces an `int` result
  - the `double` version produces a `double` result
  - etc.
- Rules for numeric operators:
  - if the operands are both of type `int`, the `int` version of the operator is used.
    - examples:
```
15 + 30
1 / 2
25 * quarters
```
  - if at least one of the operands is of type `double`, the `double` version of the operator is used.
    - examples:
```
15.5 + 30.1
1 / 2.0
25.0 * quarters
```

Incorrect Extended Change-Adder Program

```
/*
 * ChangeAdder2.java
 * Dave Sullivan (dgs@cs.bu.edu)
 * This program determines the value of some coins.
 */

public class ChangeAdder2 {
    public static void main(String[] args) {
        int quarters = 10;
        int dimes = 3;
        int nickels = 7;
        int pennies = 6;
        int cents;

        // compute and print the total value
        cents = 25*quarters + 10*dimes + 5*nickels + pennies;
        System.out.print("total in cents is: ");
        System.out.println(cents);
        double dollars = cents / 100;
        System.out.print("total in dollars is: ");
        System.out.println(dollars);
    }
}
```

Two Types of Division

- The `int` version of the `/` operator performs *integer division*, which discards the fractional part of the result (i.e., everything after the decimal).
  - examples:

expression	value
`5 / 3`	`1`
`11 / 5`	`2`

- The `double` version of the `/` operator performs *floating-point division*, which keeps the fractional part.
  - examples:

expression	value
`5.0 / 3.0`	`1.6666666666666667`
`11 / 5.0`	`2.2`

How Can We Fix Our Program?

```
/*
 * ChangeAdder2.java
 * Dave Sullivan (dgs@cs.bu.edu)
 * This program determines the value of some coins.
 */

public class ChangeAdder2 {
    public static void main(String[] args) {
        int quarters = 10;
        int dimes = 3;
        int nickels = 7;
        int pennies = 6;
        int cents;

        // compute and print the total value
        cents = 25*quarters + 10*dimes + 5*nickels + pennies;
        System.out.print("total in cents is: ");
        System.out.println(cents);
        double dollars = cents / 100;
        System.out.print("total in dollars is: ");
        System.out.println(dollars);
    }
}
```

String Concatenation

- The meaning of the + operator depends on the types of the operands.
- When at least one of the operands is a string, the + operator performs string concatenation.
  - combines two or more strings into a single string
  - example:

```
System.out.println("hello " + "world");
```

    is equivalent to

```
System.out.println("hello world");
```

String Concatenation (cont.)

- If one operand is a string and the other is a number, the number is converted to a string and then concatenated.
  - example: instead of writing

```
System.out.print("total in cents: ");
System.out.println(cents);
```

    we can write

```
System.out.println("total in cents: " + cents);
```

- Here's how the evaluation occurs:

```
int cents = 321;
System.out.println("total in cents: " + cents);
                   "total in cents: " + 321
                   "total in cents: " + "321"
                   "total in cents: 321"
```

Change-Adder Using String Concatenation

```
/*
 * ChangeAdder2.java
 * Dave Sullivan (dgs@cs.bu.edu)
 * This program determines the value of some coins.
 */

public class ChangeAdder2 {
    public static void main(String[] args) {
        int quarters = 10;
        int dimes = 3;
        int nickels = 7;
        int pennies = 6;
        int cents;

        // compute and print the total value
        cents = 25*quarters + 10*dimes + 5*nickels + pennies;
        System.out.println("total in cents is: " + cents);
        double dollars = cents / 100.0;
        System.out.println("total in dollars is: " +
          dollars);
    }
}
```

An Incorrect Program for Computing a Grade

```
/*
 * ComputeGrade.java
 * Dave Sullivan (dgs@cs.bu.edu)
 * This program computes a grade as a percentage.
 */

public class ComputeGrade {
    public static void main(String[] args) {
        int pointsEarned = 13;
        int possiblePoints = 15;

        // compute and print the grade as a percentage
        double grade;
        grade = pointsEarned / possiblePoints * 100;
        System.out.println("The grade is: " + grade);
    }
}
```

- What is the output?

Will This Fix Things?

```
/*
 * ComputeGrade.java
 * Dave Sullivan (dgs@cs.bu.edu)
 * This program computes a grade as a percentage.
 */

public class ComputeGrade {
    public static void main(String[] args) {
        int pointsEarned = 13;
        int possiblePoints = 15;

        // compute and print the grade as a percentage
        double grade;
        grade = pointsEarned / possiblePoints * 100.0;
        System.out.println("The grade is: " + grade);
    }
}
```

Type Casts

- To compute the percentage, we need to tell Java to treat at least one of the operands as a `double`.
- We do so by performing a *type cast:*

```
grade = (double)pointsEarned / possiblePoints * 100;
```

or

```
grade = pointsEarned / (double)possiblePoints * 100;
```

- General syntax for a type cast:

  (*type*) *variable*

Corrected Program for Computing a Grade

```
/*
 * ComputeGrade.java
 * Dave Sullivan (dgs@cs.bu.edu)
 * This program computes a grade as a percentage.
 */

public class ComputeGrade {
    public static void main(String[] args) {
        int pointsEarned = 13;
        int possiblePoints = 15;

        // compute and print the grade as a percentage
        double grade;
        grade = (double)pointsEarned / possiblePoints * 100;
        System.out.println("The grade is: " + grade);
    }
}
```

Evaluating a Type Cast

- Example of evaluating a type cast:

```
pointsEarned = 13;
possiblePoints = 15;
grade = (double)pointsEarned / possiblePoints * 100;
        (double)13 / 15 * 100;
              13.0 / 15 * 100;
        0.8666666666666667 * 100;
        86.66666666666667;
```

- Note that the type cast occurs *after* the variable is replaced by its value.
- It does *not* change the value that is actually stored in the variable.
  - in the example above, `pointsEarned` is still 13

Type Conversions

- Java will automatically convert values from one type to another *provided there is no potential loss of information.*
- Example: we can perform the following assignment without a type cast:

```
double d = 3;
```

variable of type `double` — value of type `int`

  - the JVM will convert the integer value `3` to the floating-point value `3.0` and assign that value to `d`
  - *any* `int` can be assigned to a `double` without losing any information

Type Conversions (cont.)

- The compiler will complain if the necessary type conversion could (at least in some cases) lead to a loss of information:

```
int i = 7.5;    // won't compile
```

variable of type `int` — value of type `double`

- This is true regardless of the actual value being converted:

```
int i = 5.0;    // won't compile
```

- To make the compiler happy in such cases, we need to use a type cast:

```
double area = 5.7;
int approximateArea = (int)area;
System.out.println(approximateArea);
```

  - what would the output be?

Type Conversions (cont.)

- When an automatic type conversion is performed as part of an assignment, the conversion happens after the evaluation of the expression to the right of the =.
- Example:

```
double d = 1 / 3;
       = 0;     // uses integer division. why?
       = 0.0;
```

A Block of Code

- A *block* of code is a set of statements that is treated as a single unit.
- In Java, a block is typically surrounded by curly braces.
- Examples:
  - each class is a block
  - each method is a block

```
public class MyProgram {
    public static void main(String[] args) {
        int i = 5;
        System.out.println(i * 3);
        int j = 10;
        System.out.println(j / i);
    }
}
```

Variable Scope

- The *scope* of a variable is the portion of a program in which the variable can be used.
- By default, the scope of a variable in Java:
  - begins at the point at which it is declared
  - ends at the end of the innermost block that encloses the declaration

```
public class MyProgram2 {
    public static void main(String[] args) {
        System.out.println("Welcome!");
        System.out.println("Let's do some math!");
        int j = 10;
        System.out.println(j / 5);
    }
}
```

scope of j

- Because of these rules, a variable cannot be used outside of the block in which it is declared.

Another Example

```
public class MyProgram3 {
    public static void method1() {
        int i = 5;
        System.out.println(i * 3);
        int j = 10;
        System.out.println(j / i);
    }

    public static void main(String[] args) {
        // The following line won't compile.
        System.out.println(i + j);

        int i = 4;
        System.out.println(i * 6);
        method1();
    }
}
```

scope of j

scope of method1's version of i

scope of main's version of i

Local Variables vs. Global Variables

```
public class MyProgram {
    static int x = 10;    // a global variable

    public static void method1() {
        int y = 5;        // a local variable
        System.out.println(x + y);
        ...
```

- Variables that are declared inside a method are *local variables.*
  - they cannot be used outside that method.
- In theory, we can define *global variables* that are available throughout the program.
  - they are declared outside of any method, using the keyword `static`
- However, we generally avoid global variables.
  - can lead to problems in which one method accidentally affects the behavior of another method

Yet Another Change-Adder Program!

- Let's change it to print the result in dollars and cents.
  - 321 cents should print as 3 dollars, 21 cents

```
public class ChangeAdder3 {
    public static void main(String[] args) {
        int quarters = 10;
        int dimes = 3;
        int nickels = 7;
        int pennies = 6;
        int dollars, cents;

        cents = 25*quarters + 10*dimes + 5*nickels + pennies;

        // what should go here?



        System.out.println("dollars = " + dollars);
        System.out.println("cents = " + cents);
    }
}
```

Conditional Execution: Deciding What to Do

- What if the user has 121 cents?
  - will print as 1 dollars, 21 cents
  - would like it to print as 1 dollar, 21 cents
- We need a means of deciding what to print at runtime.
  - known as *conditional execution*
  - the flow of control depends on a *condition* or *test.*
- Here's an example of how it would work:

```
System.out.print(dollars);
if (dollars == 1) {
    System.out.print(" dollar, ");
} else {
    System.out.print(" dollars, ");
}
// code for printing cents goes here
```

Simple Conditional Execution in Java

```
if (condition) {
    true block
} else {
    false block
}
```

```
if (condition) {
    true block
}
```

- If the condition is true:
  - the statement(s) in the true block are executed
  - the statement(s) in the false block (if any) are skipped
- If the condition is false:
  - the statement(s) in the false block (if any) are executed
  - the statement(s) in the true block are skipped

Expressing Simple Conditions

- Java provides a set of operators called *relational operators* for expressing simple conditions:

operator	name	examples
<	less than	`5 < 10` `num < 0`
>	greater than	`40 > 60` (which is false!) `count > 10`
<=	less than or equal to	`average <= 85.8`
>=	greater than or equal to	`temp >= 32`
==	equal to *(don't confuse with =)*	`sum == 10` `firstChar == 'P'`
!=	not equal to	`age != myAge`

Change Adder With Conditional Execution

```
public class ChangeAdder3 {
    public static void main(String[] args) {
        ...

        System.out.print(dollars);
        if (dollars == 1) {
            System.out.print(" dollar, ");
        } else {
            System.out.print(" dollars, ");
        }

        // Add statements to correctly print cents.
        // Try to use only an if, not an else.









    }
}
```

Classifying Bugs

- Syntax errors
  - found by the compiler
  - occur when code doesn't follow the rules of the programming language
  - examples?

Classifying Bugs

- Syntax errors
  - found by the compiler
  - occur when code doesn't follow the rules of the programming language
  - examples?
- Logic errors
  - the code compiles, but it doesn’t do what you intended it to do
  - may or may not cause the program to crash
    - called *runtime errors* if the program crashes
  - often harder to find!

Common Syntax Errors Involving Variables

- Failing to declare the type of the variable.
- Failing to initialize a variable before you use it:

```
int radius;
double area = 3.1416 * radius * radius;
```

- Trying to declare a variable when there is already a variable with that same name in the current scope:

```
int val1 = 10;
System.out.print(val1 * 2);
int val1 = 20;
```

Will This Compile?

```
public class ChangeAdder {
    public static void main(String[] args) {
        ...
        int cents;
        cents = 25*quarters + 10*dimes + 5*nickels + pennies;

        if (cents % 100 == 0) {
            int dollars = cents / 100;
            System.out.println(dollars + " dollars");
        } else {
            int dollars = cents / 100;
            cents = dollars % 100;
            System.out.println(dollars + " dollars and "
              + cents + " cents");
        }
    }
}
```

Representing Integers

- Like all values in a computer, integers are stored as binary numbers – sequences of *bits* (0s and 1s).
- With n bits, we can represent 2^n different values.
  - examples:
    - 2 bits give $2^2 = 4$ different values
      00, 01, 10, 11
    - 3 bits give $2^3 = 8$ different values
      000, 001, 010, 011, 100, 101, 110, 111
- When we allow for negative integers (which Java does) n bits can represent any integer from -2^{n-1} to $2^{n-1} - 1$.
  - there's one fewer positive value to make room for 0

Java's Integer Types

- Java's actually has four primitive types for integers, all of which represent signed integers.

type	# of bits	range of values
`byte`	8	-2^7 to $2^7 - 1$ (–128 to 127)
`short`	16	-2^{15} to $2^{15} - 1$ (–32768 to 32767)
`int`	32	-2^{31} to $2^{31} - 1$ (approx. +/–2 billion)
`long`	64	-2^{63} to $2^{63} - 1$

- We typically use `int`, unless there's a good reason not to.

Java's Floating-Point Types

- Java has two primitive types for floating-point numbers:

type	# of bits	approx. range	approx. precision
`float`	32	$+/-10^{-45}$ to $+/-10^{38}$	7 decimal digits
`double`	64	$+/-10^{-324}$ to $+/-10^{308}$	15 decimal digits

- We typically use `double` because of its greater precision.

Binary to Decimal

- Number the bits from right to left
  - example:

0	1	0	1	1	1	0	1
b7	b6	b5	b4	b3	b2	b1	b0

- For each bit that is 1, add 2^n, where n = the bit number
  - example:

0	1	0	1	1	1	0	1
b7	b6	b5	b4	b3	b2	b1	b0

decimal value = $2^6 + 2^4 + 2^3 + 2^2 + 2^0$

$64 + 16 + 8 + 4 + 1 = 93$

  - another example: what is the integer represented by 01001011?

Decimal to Binary

- Go in the reverse direction: determine which powers of 2 need to be added together to produce the decimal number.
  - example: $42 = 32 + 8 + 2$
    $= 2^5 + 2^3 + 2^1$
  - thus, bits 5, 3, and 1 are all 1s: $42 = 00101010$

- Start with the largest power of 2 less than or equal to the number, and work down from there.
  - example: what is 21 in binary?

    16 is the largest power of 2 <= 21: $21 = 16 + 5$

    now, break the 5 into powers of 2: $21 = 16 + 4 + 1$

    1 is a power of 2 (2^0), so we're done: $21 = 16 + 4 + 1$
    $= 2^4 + 2^2 + 2^0$
    $= 00010101$

Decimal to Binary (cont.)

- Another example: what is 90 in binary?

printf: Formatted Output

- When printing a decimal number, you may want to limit yourself to a certain number of places after the decimal.
- You can do so using the `System.out.printf` method.
  - example:

    ```
    System.out.printf("%.2f", 1.0/3);
    ```

    will print

    ```
    0.33
    ```

  - the number after the decimal point in the first parameter indicates how many places after the decimal should be used
- There are other types of formatting that can also be performed using this method.
  - docs.oracle.com/javase/tutorial/java/data/numberformat.html

Review

- Consider the following code fragments

```
1)  1000
2)  10 * 5
3)  System.out.println("Hello");
4)  hello
5)  num1 = 5;
6)  2*width + 2*length
7)  main
```

- Which of them are examples of:
  - literals?
  - identifiers?
  - expressions?
  - statements?