Statements

- Variable declaration

```
type variable;
```

- Assignment

```
variable = expression;
```

- Conditionals

```
if (cond) { statements; }
```

and

```
if (cond) { stmts1; } else { stmts2; }
```

- Loops

- Method invocation (aka method call)

```
objectreference.methodname(parameters);
```

or

```
classname.methodname(parameters);
```

Objects and Classes

- Defining a class:

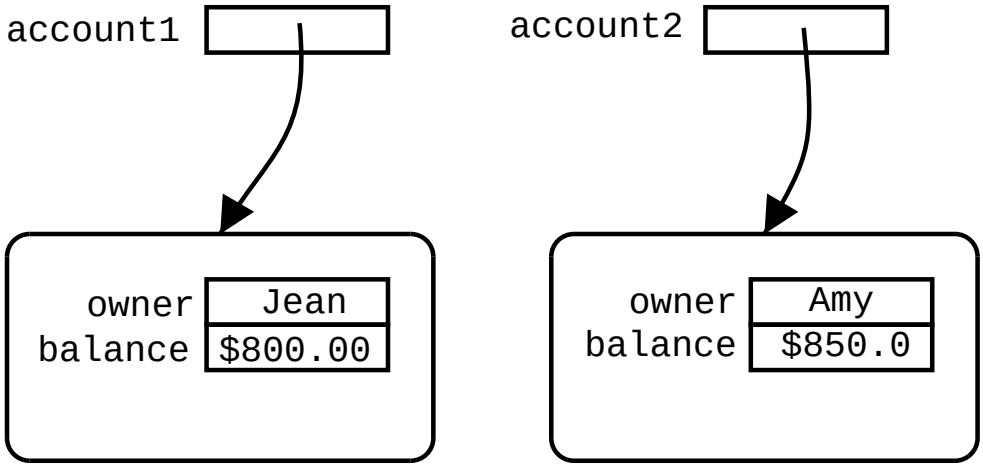
```
public class BankAccount
{
    String owner;
    double balance;

    void withdraw(double amount)
    {
      // ...
    }

    void deposit(double amount)
    {
      // ...
    }
}
```

- Note: only one class in a program has a `main` method

Objects and Classes



Classes and Objects

- Declare a variable:

```
BankAccount account1;
```

- To *create objects* we use the `new` operator (with assignment)

```
account1 = new BankAccount();
```

- To *apply operations to objects* we use the *dot* operator:

```
account1.deposit(200.00);
```

- You cannot apply methods without first creating objects

Objects and Classes

```
public class Test
{
    public static void main(String[] args)
    {
        BankAccount account1;
        account1 = new BankAccount();
        account1.deposit(200.0);
        BankAccount account2;
        account2 = new BankAccount();
        account2.deposit(150.0);
    }
}
```

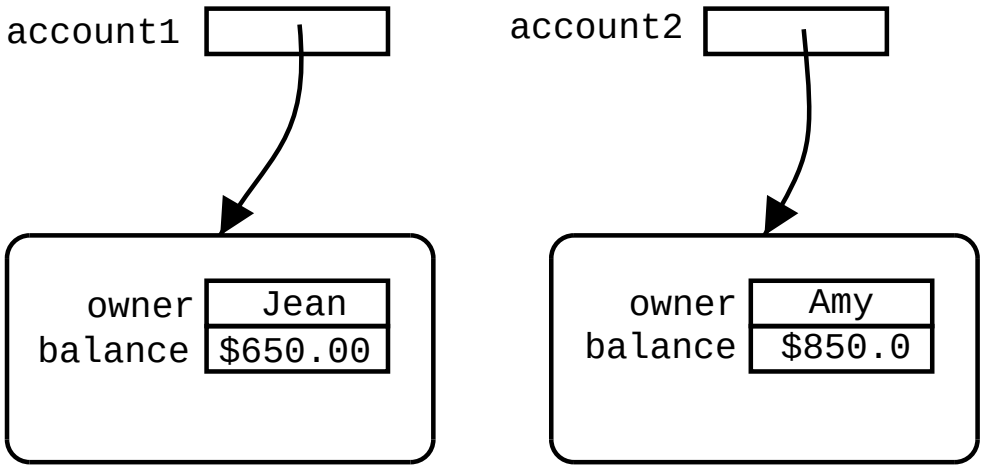
- Each object has its own separate *identity*, its own individual *state*

Objects and Classes

```
account1 . withdraw (150.00);
```

object method parameters

- Applying a method to an object affects only the object it is being applied to.




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

object method parameters

Scanner, Classes and Objects

```
int n;
Scanner myScanner;
myScanner = new Scanner(System.in);
n = myScanner.nextInt();
```

Strings

```
String greetings;
greetings = “Hello”;
```

Strings

```
String greetings;
greetings = new String(“Hello”);
```

Strings, Classes and Objects

"	b	o	n	j	o	u	r	"
	0	1	2	3	4	5	6	

- In strings,
  - the first character has index 0
  - the second character has index 1
  - the third character has index 2
  - ...
  - the last character has index l-1, where l is the length of the string

Strings

```
String greetings;
greetings = new String(“Hello”);

int n;
n = greetings.length();

char letter;
letter = greetings.charAt(2);

if (greetings.equals(“hello”))
{
    System.out.println(letter);
}
```

The Random class

- Random number generation
- Random class methods

```
int nextInt()
int nextInt(int max)
float nextFloat()
double nextDouble()
```

- Must import the class from the `java.util` package

The Random class

```
import java.util.Random;
public class Test
{
  public static void main(String[] args)
  {
    Random generator;
    generator = new Random();
    double x;
    x = generator.nextDouble();
    x = x * 5.0;
  }
}
```

Formatting numbers

- `NumberFormat` and `DecimalFormat` classes from the `java.text` package

- Methods

```
DecimalFormat(String pattern)
String format(double number)
void applyPattern(String pattern)
```

Formatting numbers

```
import java.text.DecimalFormat;
public class Test
{
  public static void main(String[] args)
  {
    double n = 1.618314141;
    String output = "";
    DecimalFormat formatter;
    formatter = new DecimalFormat("0.##");
    output = formatter.format(n);
    System.out.println(output);
  }
}
```

Classes and Objects

- Declare a variable:

```
ClassName variable;
```

- To *create objects* we use the `new` operator (with assignment)

```
variable = new ClassName();
```

- To *apply operations to objects* we use the *dot* operator:

```
variable.methodname(parameters);
```

- You cannot apply methods without first creating objects

Static methods

- So far, all method calls that we have used take the form

  objectreference.*methodname*(*parameters*)

- But there are some methods that take the form

  ClassName.*methodname*(*parameters*)

- These are called *static methods*

- Static methods do not represent operations on objects, but services provided by a class

- For example:

```
x = Math.sqrt(3);
```

Static methods and class libraries

```
double cathetus1, cathetus2, hypothenuse;
cathetus1 = 3.0;
cathetus2 = 4.0;
hypothenuse = Math.sqrt( Math.pow( cathetus1, 2 ) +
                         Math.pow( cathetus2, 2 ) );
```

Static methods and class libraries

The Math class has many useful static methods, such as:

Method	Description
`static int abs(int num)`	returns the absolute value of `num`
`static double pow(double num, double power)`	returns $\mathbf{num}^{\mathbf{power}}$
`static double sqrt(double num)`	returns $\sqrt{\mathbf{num}}$
`static double sin(double angle)`	returns $\sin(\mathbf{angle})$
`static double cos(double angle)`	returns $\cos(\mathbf{angle})$
`static double tan(double angle)`	returns $\tan(\mathbf{angle})$
`static double floor(double num)`	returns the largest integer less or
`static double ceil(double num)`	returns the smallest integer greate

Some shortcuts

```
x++;
```

means

```
x = x + 1;
```

```
x--;
```

means

```
x = x - 1;
```

```
x += 3;
```

means

```
x = x + 3;
```

Some shortcuts

- ++ and -- can be used inside arithmetic expressions (but it is not recommendable)

```
x = y-- * 2;
```

means:

```
x = y * 2;
y = y - 1;
```

and

```
x = --y * 2;
```

means

```
y = y - 1;
x = y * 2;
```

Some syntactic shortcuts

- The ++ and -- operators can be used within expressions (but they shouldn't)

```
v = 3;
if (v++ >= 4) System.out.println("A");
```

is not the same as

```
v = 3;
if (++v >= 4) System.out.println("A");
```

Some syntactic shortcuts

- The ++ and -- operators affect evaluation of conditions

```
v = 4;
if (v++ >= 4 && v < 5)    System.out.println(“A”)
```

is not the same as

```
v = 4;
if (v < 5 && v++ >= 4)    System.out.println(“A”)
```

Characters

- Values of the `char` data type can be compared using the traditional relational operators:

```
char a = 'P', b = 'Q';
boolean c, d, e, f, g, h;
c = a == b;     // c == false
d = a != b;     // d == true
e = a < b;      // e == true
f = a > b;      // f == false
g = a <= b;     // g == true
h = a >= b;     // h == false

char a = 'Q', b = 'Q';
boolean c, d, e, f, g, h;
c = a == b;     // c == true
d = a != b;     // d == false
e = a < b;      // e == false
f = a > b;      // f == false
g = a <= b;     // g == true
h = a >= b;     // h == true
```

Data conversion

- Sometimes it is useful to look at data as if they were from a different type
- For example:
  - Adding an integer and a double
  - Obtaining the ASCII code of a character
- Forms of data conversion:
  - Implicit:
    * Assignment conversion
    * Promotion
  - Explicit: Casting

Primitive Data Types

General category	Type	Description	Exar
Numeric	int	Integers	0,1
	long	Long integers	65
	short	Short integers	2,-
	byte	Bytes	25
	float	Rationals	1.3
	double	Rationals	1.6
Text	char	Single characters	'x'
	String	Sequences of characters	"ab
Logic	boolean	Truth values	true

Data conversion

- Assignment conversion: A value of one type is assigned to a variable of a different type, as long as the types are compatible

```
int n = 7;
double d = n;
long k = n;
int m = d; // Wrong: compile-time error
```

- Promotion: an expression "promotes" the types of its operands to its "largest" type

```
int m = 8;
float x = 3.0f, y;
y = x + m;
```

Data conversion

- Casting expressions (not a statement)

  (*type*)*expression*

- Examples:

```
int n = 3;
double p;
p = (double)n + 4.0;

int a = 3, b = 8;
float c, d;
c = b/a;
d = (float)b/a;
System.out.println(c);  // 2.0
System.out.println(d);  // 2.666666...
```

Data conversion

```
double r = 2.41;
int a;
a = r; // Error
```

Data conversion

```
double r = 2.41;
int a;
a = (int)r;    //OK:  Narrowing casting
```

Data conversion

- There are two types of casting:
  - Narrowing conversions: from a type which requires more memory to a type that requires less
  - Widening conversions: from a type which requires less memory to a type which requires more

- If `expression` has type `t`, and `t` requires more memory than type `s`, then `(s)expression` is a narrowing conversion (e.g. `int` to `byte`, `double` to `float`, `float` to `int`, ...)

- If `expression` has type `t`, and `t` requires less memory than type `s`, then `(s)expression` is a widening conversion (e.g. `byte` to `double`, `long` to `int`, ...)

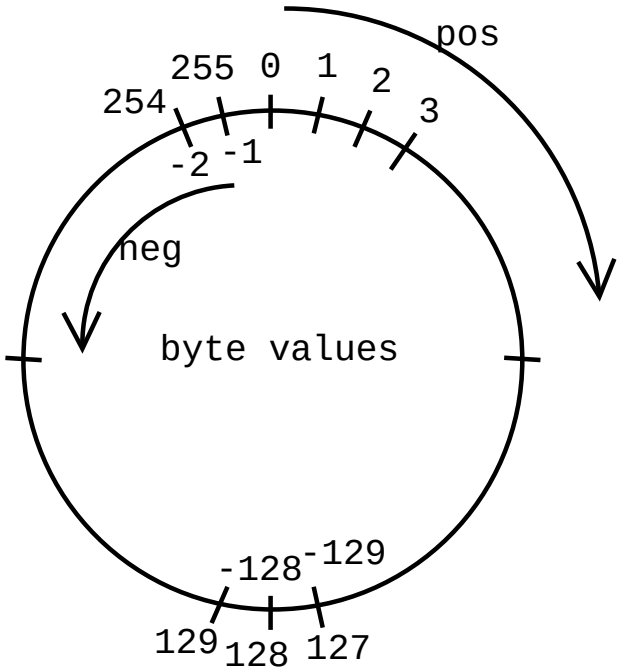
Data conversion

- Widening conversions are safe: no loss of information
- Narrowing conversions are not safe: possible loss of information

```
float x = 2.71f;
int i = (int)x;
// i == 2

int k = 130;
byte b = (byte)k;
// b = -126
```

Data conversion



128 = -128
129 = -127
256 = 0
257 = 1

byte b
int i
k is any integer

b+$k2^8$=b
i+$k2^{32}$=i

Statements

- Variable declaration

```
type variable;
```

- Assignment

```
variable = expression;
```

- Conditionals

```
if (cond) { statements; }
```

and

```
if (cond) { stmts1; } else { stmts2; }
```

- Loops
- Method invocation (aka method call)

  ```
  objectreference.methodname(parameters);
  ```

  or

  ```
  classname.methodname(parameters);
  ```

Statements

- The role of statements

Statement	Role
Assignment	to change the value of a variable
Conditionals	to make decisions
Method calls	to send a message to an object, to ask an object to perform an action
Static method calls	to execute a procedure
Loops	to repeat some action(s) several times

Loops

- The loop is a statement used to describe a task which is *repetitive*

- For example: print the first 100 odd integers

```
System.out.println(1);
System.out.println(3);
System.out.println(5);
System.out.println(7);
System.out.println(9);
System.out.println(11);
System.out.println(13);
//...
```

- What if we want to print the first 1000 odd numbers?

- What if the user is supposed to give the program the number of odd numbers?

Loops

- The basic loop statement:

```
while (boolean_expression) {
  list_of_statements;
}
```

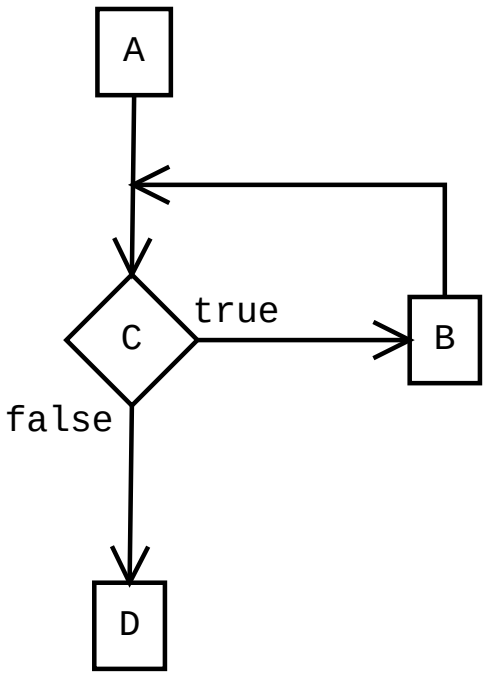
- Semantics: the execution of a while loop proceeds as follows:

1. The boolean expression is evaluated
   (a) If it is false,
      i. the loop stops
      ii. and computation proceeds directly after the loop
   (b) If it is true,
      i. the list of statements is executed,
      ii. and when finished, the whole process is repeated from step 1

Loops

```
A;
while (C) {
  B;
}
D
```

- Control flow diagram:



Loops

```
int counter, number;
counter = 1;
number = 1;
while (counter <= 100) {
    System.out.println(number);
    number = number + 2;
    counter++;
}
System.out.println(“Done”);
```

Loops

```
int counter, number;
counter = 1;
number = 1;
while (counter <= 3) {
    System.out.println(number);
    number = number + 2;
    counter++;
}
System.out.println("Done");
```

counter	number
-	-

(This table shows the values of the variables just before the statement in red is executed)

Printed:

Loops

```
int counter, number;
counter = 1;
number = 1;
while (counter <= 3) {
    System.out.println(number);
    number = number + 2;
    counter++;
}
System.out.println("Done");
```

counter	number
1	-

Printed:

Loops

```
int counter, number;
counter = 1;
number = 1;
while (counter <= 3) {
    System.out.println(number);
    number = number + 2;
    counter++;
}
System.out.println("Done");
```

counter	number
1	1

Printed:

Loops

```
int counter, number;
counter = 1;
number = 1;
while (counter <= 3) {
    System.out.println(number);
    number = number + 2;
    counter++;
}
System.out.println("Done");
```

counter	number
1	1

Printed:

Loops

```
int counter, number;
counter = 1;
number = 1;
while (counter <= 3) {
    System.out.println(number);
    number = number + 2;
    counter++;
}
System.out.println("Done");
```

counter	number
1	1

Printed:

1

Loops

```
int counter, number;
counter = 1;
number = 1;
while (counter <= 3) {
    System.out.println(number);
    number = number + 2;
    counter++;
}
System.out.println("Done");
```

counter	number
1	3

Printed:

1

Loops

```
int counter, number;
counter = 1;
number = 1;
while (counter <= 3) {
    System.out.println(number);
    number = number + 2;
    counter++;
}
System.out.println("Done");
```

counter	number
2	3

Printed:

1

Loops

```
int counter, number;
counter = 1;
number = 1;
while (counter <= 3) {
    System.out.println(number);
    number = number + 2;
    counter++;
}
System.out.println("Done");
```

counter	number
2	3

Printed:

1

Loops

```
int counter, number;
counter = 1;
number = 1;
while (counter <= 3) {
    System.out.println(number);
    number = number + 2;
    counter++;
}
System.out.println("Done");
```

counter	number
2	3

Printed:

1
3

Loops

```
int counter, number;
counter = 1;
number = 1;
while (counter <= 3) {
    System.out.println(number);
    number = number + 2;
    counter++;
}
System.out.println("Done");
```

counter	number
2	5

Printed:

1
3

Loops

```
int counter, number;
counter = 1;
number = 1;
while (counter <= 3) {
    System.out.println(number);
    number = number + 2;
    counter++;
}
System.out.println("Done");
```

counter	number
3	5

Printed:

1
3

Loops

```
int counter, number;
counter = 1;
number = 1;
while (counter <= 3) {
    System.out.println(number);
    number = number + 2;
    counter++;
}
System.out.println("Done");
```

counter	number
3	5

Printed:

1
3

Loops

```
int counter, number;
counter = 1;
number = 1;
while (counter <= 3) {
    System.out.println(number);
    number = number + 2;
    counter++;
}
System.out.println("Done");
```

counter	number
3	5

Printed:

1
3
5

Loops

```
int counter, number;
counter = 1;
number = 1;
while (counter <= 3) {
    System.out.println(number);
    number = number + 2;
    counter++;
}
System.out.println("Done");
```

counter	number
3	7

Printed:

1
3
5

Loops

```
int counter, number;
counter = 1;
number = 1;
while (counter <= 3) {
    System.out.println(number);
    number = number + 2;
    counter++;
}
System.out.println("Done");
```

counter	number
4	7

Printed:

1
3
5

Loops

```
int counter, number;
counter = 1;
number = 1;
while (counter <= 3) {
    System.out.println(number);
    number = number + 2;
    counter++;
}
System.out.println(“Done”);
```

counter	number
4	7

Printed:

```
1
3
5
Done
```

Loops

```
int counter = 1;
int number = 1;
while (counter <= 10000) {
    System.out.println(number);
    number = number + 2;
    counter++;
}
System.out.println(“Done”);
```

Loops

- `while` is *not* the same as `if`

```
int maximum = scanner.nextInt();
int counter = 1;
int number = 1;
while (counter <= maximum) {
    System.out.println(number);
    number = number + 2;
    counter++;
}
```

Loops

- `while` is *not* the same as `if`

```
int maximum = scanner.nextInt();
int counter = 1;
int number = 1;
if (counter <= maximum) {
    System.out.println(number);
    number = number + 2;
    counter++;
}
```

- The while statement executes a statement or list of statements *repeteadely*, until its condition becomes false

- The if statement executes a statement or list of statements *once*, and only if its condition is true

Loops

- A loop may not terminate

```
int maximum = scanner.nextInt();
int counter = 1;
int number = 1;
while (counter <= maximum) {
    System.out.println(number);
    number = number + 2;
}
```

- A loop will not terminate if its condition is always true
- The condition of a loop will remain true if its variables never change

Loops

- The variables of the condition must change in a way which eventually makes the condition false
- If the variables change, but in a way that does not make the condition false eventually, then the loop does not terminate

```
int maximum = scanner.nextInt();
int counter = 1;
int number = 1;
while (counter <= maximum) {
    System.out.println(number);
    number = number + 2;
    counter--;
}
```

Loops

- Will this terminate?

```
int i;
i = 1;
while (i != 10) {
  //...
  i = i + 2;
}
```

Loops

- Will this terminate?

```
int i;
i = 100;
while (i != 0) {
  //...
  i = i / 2;
}
```

Loops

- Will this terminate?

```
int i;
i = 10;
while (i != 3) {
  //...
  i = i / 2;
}
```

Loops

- Will this terminate?

```
double i;
i = 10;
while (i != 0) {
  //...
  i = i / 2;
}
```

Loops

- Termination is important

The end