Methods

Objectives

- Defining *methods*
- Sending messages to objects and asking them to do something
- *Compile a method to turn it into something the computer understands*
- Make methods reusable

- Methods are where the real actions take place. There are many predefined methods in Applets, but in Applications, there is only one – the main method.
- Let us look at a few things before going into creating our own methods.
- Library
  - (definition) A library is a bunch of code someone else wrote that many other programmers can use. Just like many people might use books in a real library over and over again, the classes in a library of code are likely to be used by many people over and over again. Classes in a library that perform related functions are grouped into packages.

- Package
  - (definition) A package is a group of related classes that can be used over and over again to solve a common programming problem.
  - Anyone can write a package (or a library containing several related packages) and make it available to other programmers. The folks who made Java (Sun) include many packages that are very useful with the language when you install it. Packages have names (e.g., java.awt).
  - The objects in a package can be accessed using the syntax packagename.objectname
  - If you are using an object from a package a lot (such as java.awt.Color you can add the following to the very top of your code (the first line of your .java file is the best place to put it):
    - import packagename.objectname;

- You can view the documentation for the classes in Java 7 or look for the documentation associated with your installed version of Java. For example, the documentation for java.awt.Color lists several colors you could pass in to setPenColor (Color.BLACK, Color.BLUE, Color.CYAN, Color.DARK_GRAY, Color.GRAY, Color.GREEN, Color.LIGHT_GRAY, Color.MAGENTA, Color.ORANGE, Color.PINK...).

Writing Methods

- Large programs are written by subdividing them into methods.
  - Method is a self-contained section of code belonging to the class that achieves a specific behavior for that class.
  - methods are used to simplify the complexity of the programs.
  - The main is a specialized method in the sense that is it the starting point and only gets called during execution.

Messages and Methods

- A method is a message, a request to do something. Inside a method is a block of Java statements. The method gives them a name (remember, names are important) so that we can ask the object to invoke them. A method also encodes decisions about how something should be done.
- As a programmer, one of your jobs is to create methods.

- Methods are defined *inside* a class definition like we did in the Car class:

```
public class Car {
public int year;
   public String make;
   public boolean running;

   public void isRunning(boolean on) {
      running = on;
   }
}
```

- Notice that the entire method is defined inside the class body, and that a method definition, like a class definition, consists of two parts: a *header* and a *body*.

- The syntax for declaring a method is
  - visibility ReturnType name([parameterList])
- Method header consists of these four pieces:
  - visibility: This is called the access specifier.
    - public means that objects from different classes are free to ask this object to perform this action.
    - private means that only this objects attributes and methods would be able to invoke the method.
  - returnType: Every method can, potentially, produce a single value.
    - If the method produced a value, we'd put the type of that value [number, character, etc] here. If a method produces no value, you have to say so by using a return type of void.
  - name: This is the name of the method.
  - parameterList : Every method definition has a set of parentheses following the method name. If our method required arguments, here's where we would put the definitions for those arguments or we just leave this section blank.

Common mistakes

- after the class definition (at the very end of the file), like:

```
public class Car {
   ...
}
public void go () {
}
```

- Instead, do:

```
public class Car {
    ...
    public void go () {
    }
}
```

- inside another method, like:

```
public void go () {
    ...
    public void stop () {
        ...
    }
}
```

- Instead, do:

```
public void go () {
    ...
}
public void stop () {
    ...
}
```

When writing methods

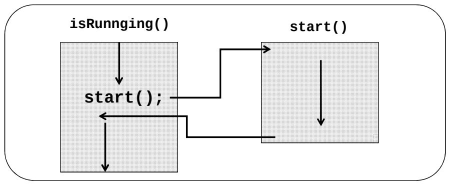
- A method declaration specifies the code that will be executed when the method is invoked (or called)
- When a method is invoked, the flow of control jumps to the method and executes its code
- When complete, the flow returns to the place where the method was called and continues
- The invocation may or may not return a value, depending on how the method was defined

Scope

- Something to keep in mind when writing code is scope. Scope defines where a variable can be used in a program. (the area in a program in which that data can be referenced).
- *Local Scope*: a variable's scope is limited to the method in which it is declared. (local data).
- *Class/Global Scope*: an instance variable can be accessed anywhere within the class instance.

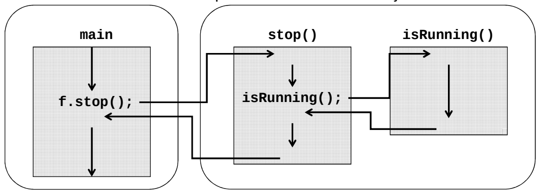
Method Control Flow

- The called method could be within the same class, in which case only the method name is needed



Method Control Flow

- The called method could be part of another class or object



Recap: Arguments and Parameters

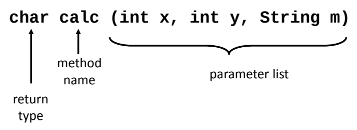
- *Arguments* refer to the values that are passed to a method.
- The arguments passed to a method must match the *formal parameters* of that method.
- *Invoking* a method involves a method call
- Defining a method involves designing a method definition.
- *Qualified names* (dot notation), are used to refer to methods within other classes.

Recap: Visibility Modifiers

- A *modifier* is a Java reserved word that specifies particular characteristics of a method or data value
- the modifier *final* is to define a constant
- Members of a class that are declared with *public* visibility can be accessed from anywhere
- Members of a class that are declared with *private* visibility can only be accessed from inside the class
- Members declared without a visibility modifier have default visibility and can be accessed by any class in the same package

Method Declaration Example

- A method declaration begins with a method header



```
char calc (int x, int y, String m)
```

- The parameter list specifies the type and name of each parameter
- The name of a parameter in the method declaration is called a formal argument

Method Declarations

- The method header is followed by the method body

```
char calc (int x, int y, String m) {
  int sum = x + y;
  char result = m.charAt (sum);
  return result;
}
```

The return expression must be consistent with the return type

- sum and result are local data.
- They are created each time the method is called, and are destroyed when the method finishes executing

The return Statement

- The *return type* of a method indicates the type of value that the method sends back to the calling location
- A method that does not return a value has a *void* return type
- The return statement specifies the value that will be returned and must conform to the return type

19

Passing a Value or a Reference

- The effect of passing an argument differs, depending on whether the argument is a reference value or a primitive value
- Values of type int, boolean, float, and double are examples of *primitive types*. A primitive argument cannot be changed by a method call.
- All objects (String, Math) are reference types. *Reference arguments* can be changed by a method call.

20

Parameters

- Each time a method is called, the actual arguments in the invocation are copied into the formal arguments

```
ch = calc (25, count, "Hello");

char calc (int x, int y, String m) {
   int sum = x + y;
   char result = m.charAt (sum);
   return result;
}
```

```
x = 25
y = count
m = "Hello"
```

21

Primitive Values

- For primitive type parameters, a copy of the argument's value is passed to the method.

```
int x = 25,  y = 5;
int avg = average(x, y);

public int average(int n, int m){
        return n / m;
}
```

- 25 is copied into n and 5 is copied into m when average() is called. So average() has no access to x and y itself and x and y remains equal to 25 and 5 even though n and m can be changed

22

Method Decomposition

- A method should be relatively small, so that it can be readily understood as a single entity
- A potentially large method should be decomposed into several smaller methods as needed for clarity
- Therefore, a service method of an object may call one or more support methods to accomplish its goal

23

Data Scope revisited

- The *scope* of data is the area in a program in which that data can be used (referenced)
- Data declared at the class level can be used by all methods in that class
- Data declared within a method can only be used in that method
- Data declared within a method is called *local data*

24

A first method

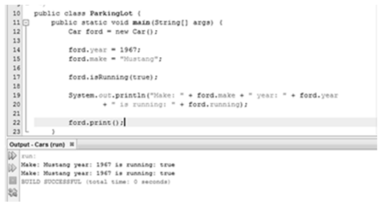
- By creating methods we also shorten the length of code. Let us look at our Car.java class again.
- Every time we create an object and want to display the information about the Car object we would have to create complicated output lines (the System.out.println).
- Why don't we create a method that displays the information for each object and all we have to do is call that method.

- In the Car.java create the following method:

```
public void print() {
        System.out.println("Make: " + make + " year: " + year
              + " is running: " + running);
    }
```

- Does this look familiar? It should, this is almost identical to the line of code in the main method of ParkingLot. However, since we are in the Car object, we don't need to refer to the attributes by object name. We have direct access to them.
- Add ford.print(); to the end of you main method and run the code.

- If you leave the previous println in you will get the same out put.



```
public class ParkingLot {
    public static void main(String[] args) {
        Car ford = new Car();

        ford.year = 1967;
        ford.make = "Mustang";

        ford.isRunning(true);

        System.out.println("Make: " + ford.make + " year: " + ford.year
                + " is running: " + ford.running);

        ford.print();
    }
```

```
run:
Make: Mustang year: 1967 is running: true
Make: Mustang year: 1967 is running: true
BUILD SUCCESSFUL (total time: 0 seconds)
```

Try by doing

- Play around a little bit with the code by creating a new Car object called Toyota. Assign some values like we did for the ford and output it using just a call to the print method.

Our Answer

```
public static void main(String[] args) {
      Car ford = new Car();
      ford.year = 1967;
      ford.make = "Mustang";
      ford.isRunning(true);
      ford.print();

      Car vw = new Car();
      vw.make = "beetle";
      vw.year = 2013;
      vw.isRunning(true);

      vw.print();
      vw.isRunning(false);
      vw.print();
 }
```

Overloading Methods

- Method overloading is the process of using the same method name for multiple methods
- The signature of each overloaded method must be unique
- The signature includes the number, type, and order of the parameters
- The compiler must be able to determine which version of the method is being invoked by analyzing the parameters
- The return type of the method is not part of the signature

Overloading Methods

VERSION 1

```
float multiply (float x) {
  return x;
}
```

VERSION 2

```
float multiply (float x, int y){
  return x*y;
}
```

```
float result = multiply (4.32f, 25);
```

Overloaded Methods

- The println method is overloaded:
  - println (String s)
  - println (int i)
  - println (double d) etc.
- The following lines invoke different versions of the println method
  - System.out.println ("The total is:");
  - System.out.println (total);

Summary

- We've learned how to create a method, using the syntax:
  - visibility returnType name([type name, type name, ...])
- Once a method is written, you can use it over and over again. An example is the method you just created, print()
- When writing code, an important goal is to implement the right method: A method that you will want to use over and over again.
- One thing that helps make methods more useful is their parameters. We learned how paremeters are specified and stored so that you can use them in your methods.
- The signature of a method consists of its name, its return type, and the number and type of its formal parameters.
- A class may NOT contain more than one method with the same signature.

- We took a moment to look at formal parameter as a place holder in a method declaration and it always consists of a type followed by variable identifier.
  - An argument is a value that is passed to a method via a formal parameter when the method is invoked.
  - A method's parameters constrain the type of information that can be passed to a method.
  - Parameter Passing: When an argument of primitive type is passed to a method, it cannot be modified within the method.
  - When an argument of reference type is passed to a method, the object it refers to can be modified within the method.
  - Except for void methods, a method invocation or method call is an expression which has a a value of a certain type.