More on Java

Object-Oriented Programming

Outline

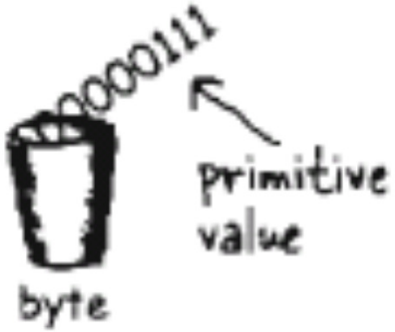
- Instance variables vs. local variables
- Primitive vs. reference types
- Object references, object equality
- Objects' and variables' lifetime
- Parameters passing and return values
- Methods overloading
- this reference
- Input/Output

- Readings:
  - HFJ: Ch. 3, 4.
  - GT: Ch. 3, 4.

Variables and types

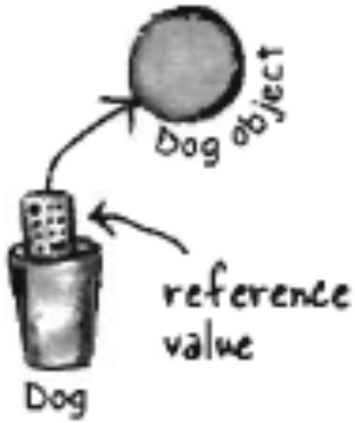
- Two kinds of variables: *primitive* and *object reference*.
- primitive variables hold fundamental types of values: int, float, char…(*)

```
byte a = 7;
boolean done = false;
```



- reference variables hold *references* to objects (similar to pointers)

```
Dog d = new Dog();
d.name = "Bruno";
d.bark();
```



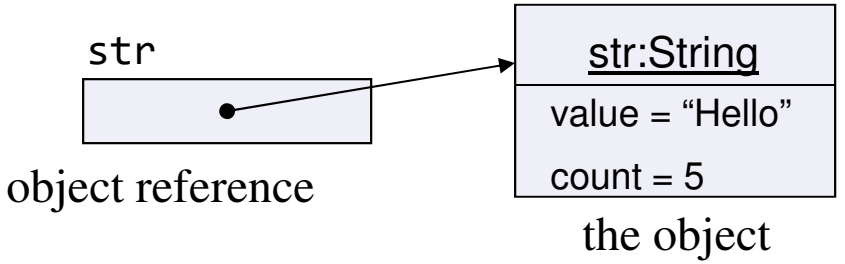
(*) read textbook

Primitive data types

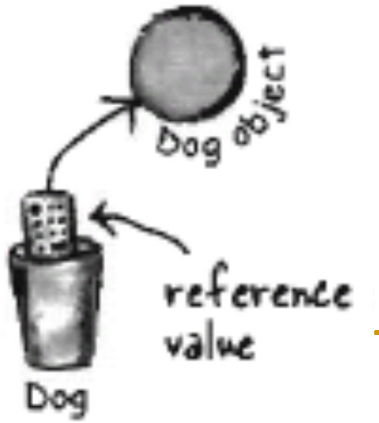
- Java's primitive types:
  - Numerical: byte, int, long, float, double
  - Logical: boolean (true/false)
  - Characters: char
- Primitive data are NOT objects
- There're corresponding wrapper classes, useful when we want to treat primitive values as objects
  - Integer, Float, …
    - Integer count = new Integer(0);
  - Provide utility functions: parseInt(), valueOf()…

Object references – controlling objects

```
str = new String("Hello");
```



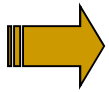
- There is actually no such thing as an *object variable*.
- There‘re only ***object reference variables***.
- An object reference variable represents a way to access an object, something like a pointer.
- Think of an object reference as a remote control



Object equality

- "==" and "!=" compares references (not objects) to see if they are refering to the same object.

```
Integer b = new Integer(10);
Integer c = new Integer(10);
Integer a = b;
```



```
a==b is true
b==c is false
```

- Use the `equals()` method to see if two objects are equal.

```
Integer b = new Integer(10);
Integer c = new Integer(10);

if (b.equals(c)) { // true };
```

Object equality

Method `equals()`

- Pre-defined classes:
  - Ready to use

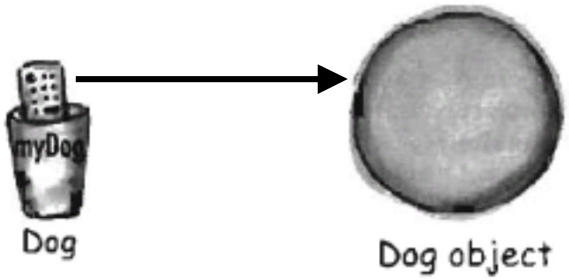
```
Integer m1 = new Integer(10);
Integer m2 = new Integer(10);
System.out.print(m1.equals(m2));
```

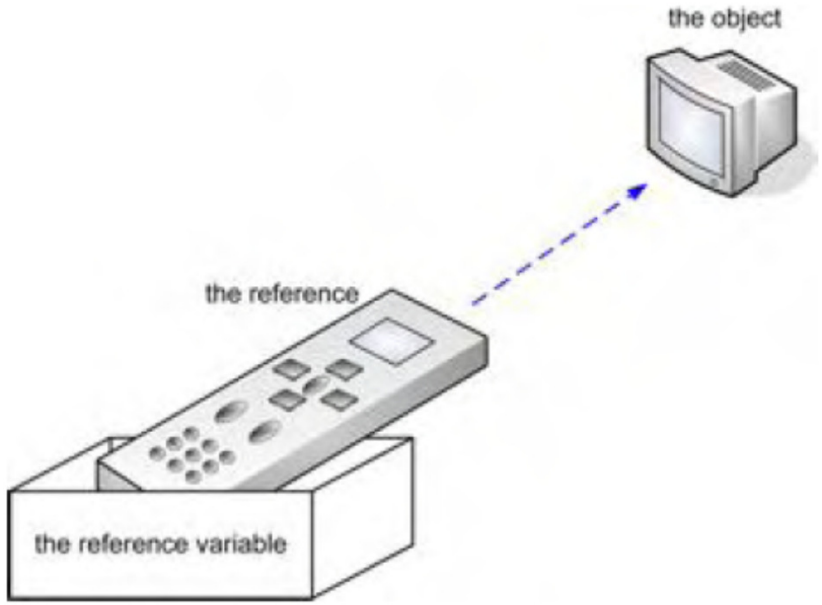
- User-created classes:
  - equals() must be defined, otherwise, it always returns *false*
  - This is overriding (more on that later)

```
class MyInteger {
    private int value;
    public boolean equals (MyInteger other) {
        return (value == other.value);
    }
...
}
```

Object references

```
Dog myDog = new Dog();
```



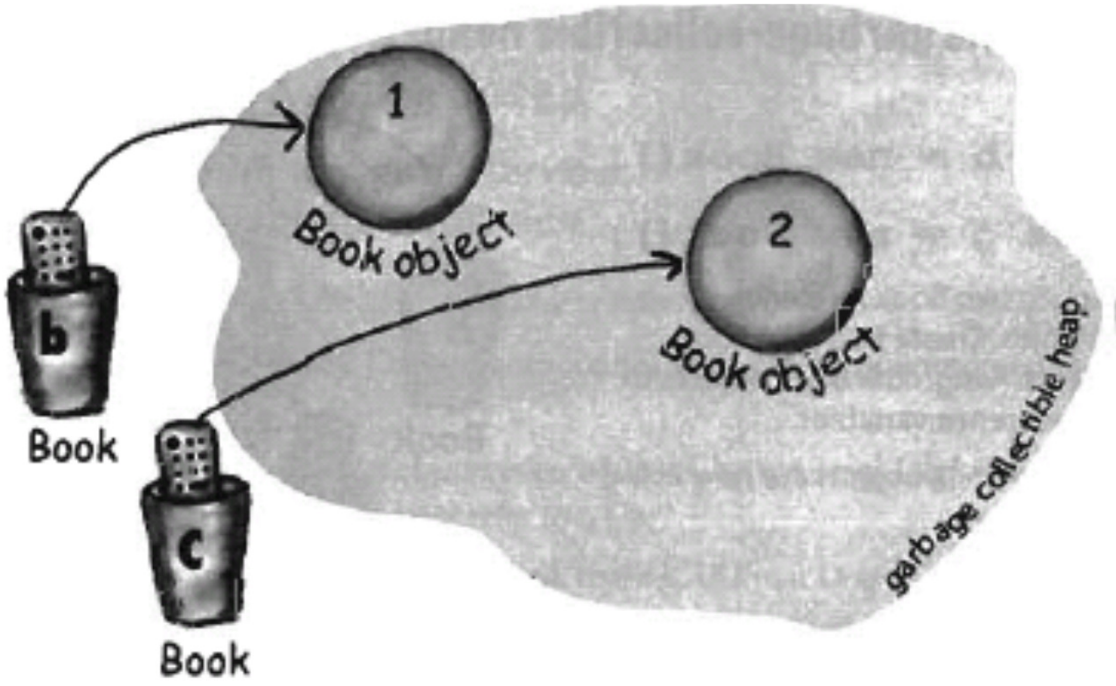


Remember: **References are not objects!**

Object's life on the heap

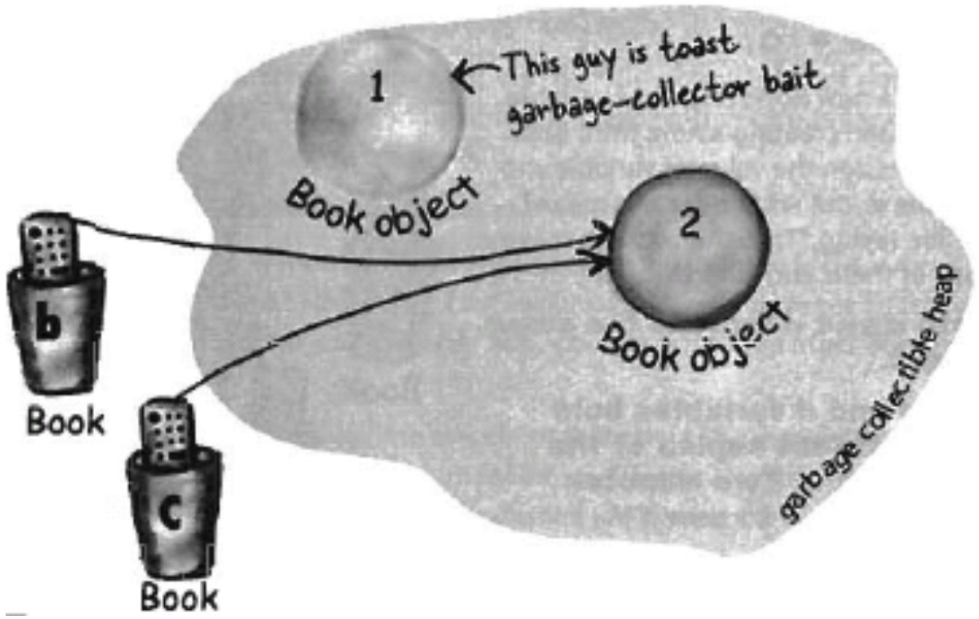
- Objects are created in the heap memory
  - a constructor is automatically called to initialize it
  - the set of parameters determine which constructor to call and the initial value of the object

```
Book b = new Book();
Book c =
  new Book("Harry Potter");
```



Object's life on the heap

when an object is no longer used,
i.e. there's no more reference to it, it will be collected and freed by Java garbage collector.



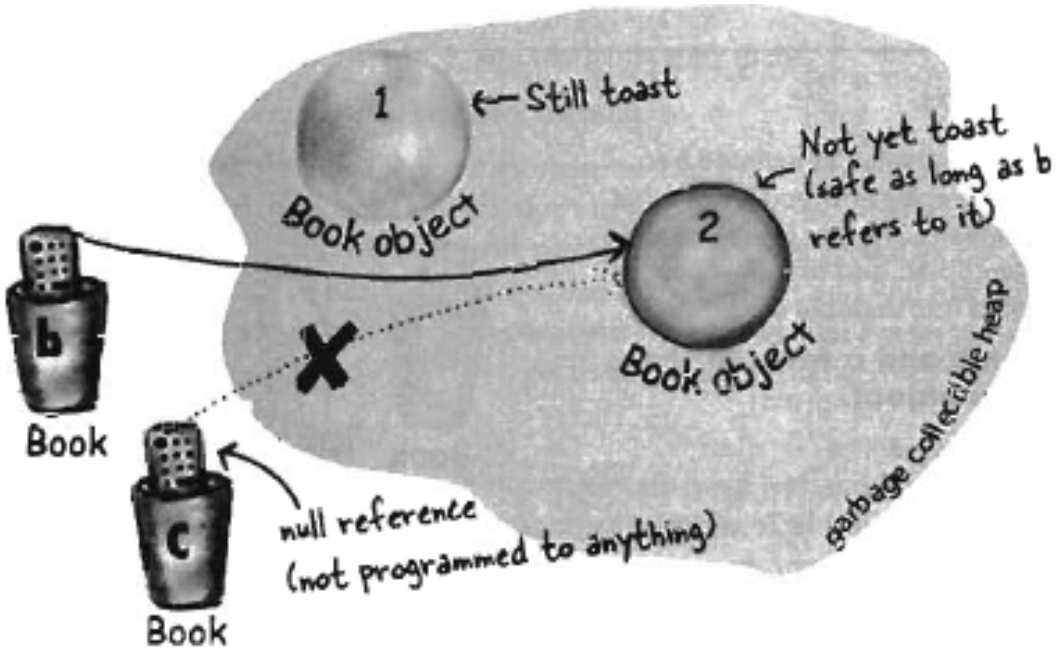

```
Book b = new Book();
Book c = new Book();
b = c;
```

There is no way to reach Book object 1.

It is ready to be collected.

Object's life on the heap

```
Book b = new Book();
Book c = new Book();
b = c;
c = null;
```



Book object 1 is waiting to be disallocated.

Book object 2 is safe as b is still referring to it.

Garbage collection

- To reclaim the memory occupied by objects that are no longer in use
- Programmers don't have to disallocate objects
- Java Virtual Machine (JVM) performs automatic garbage collection
  - Method finalize() is called by JVM, not programmers.
  - Guarantee no memory leaks
- However, there's no guarantee when/whether an object is freed before the program terminates
  - Might not needed as memory is still available
  - Clean-up tasks must be done explicitly by other "clean-up" methods rather than finalize()

Instance variables vs. local variables

Instance variables

- belong to an object
- located inside the object in the heap memory
- has the same lifetime as the object

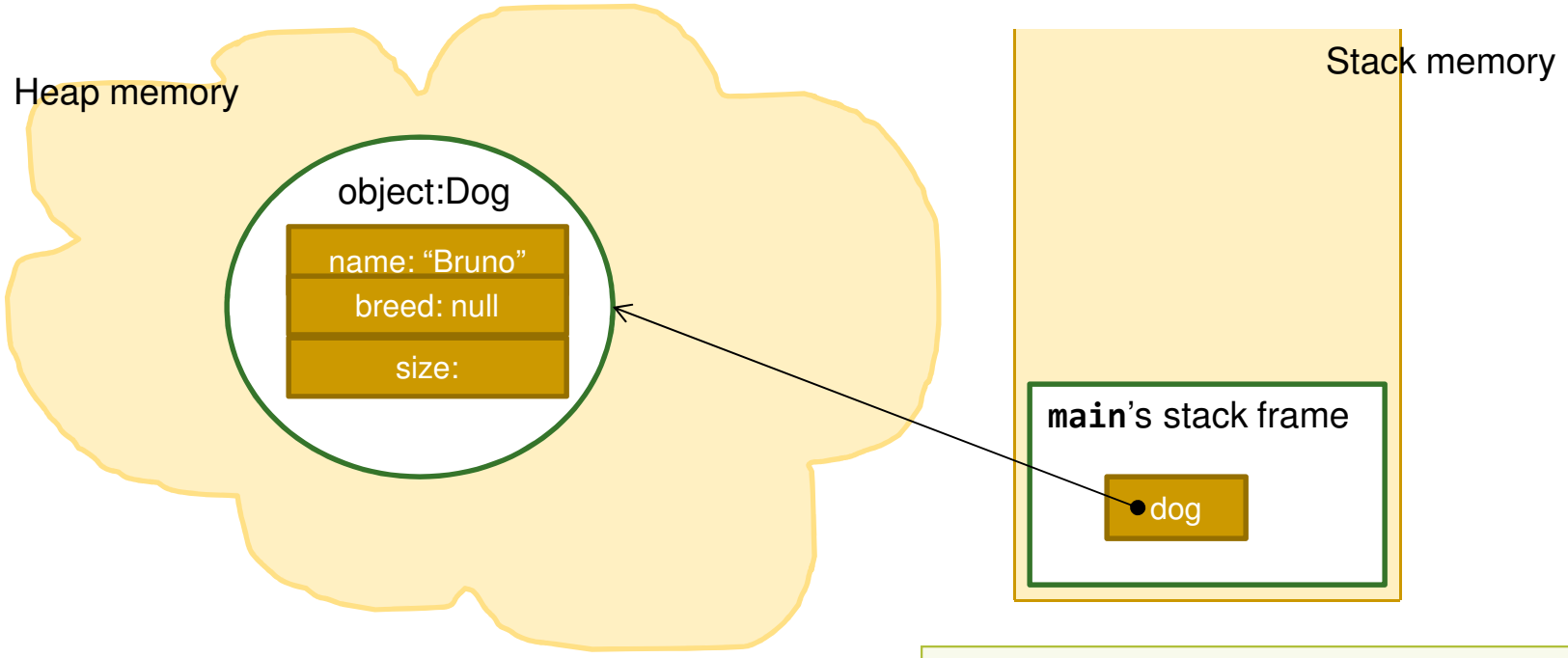
Local variables

- belong to a method
- located inside the method's frame in the stack memory
- has the same lifetime as the method call.

```
class Dog {
  int size;
  String breed;
  String name;
  ...
}
```

```
public class DogTestDrive {
  public static void main(String [
    Dog dog = new Dog();
    dog.name = "Bruno";
    dog.bark();
  }
}
```

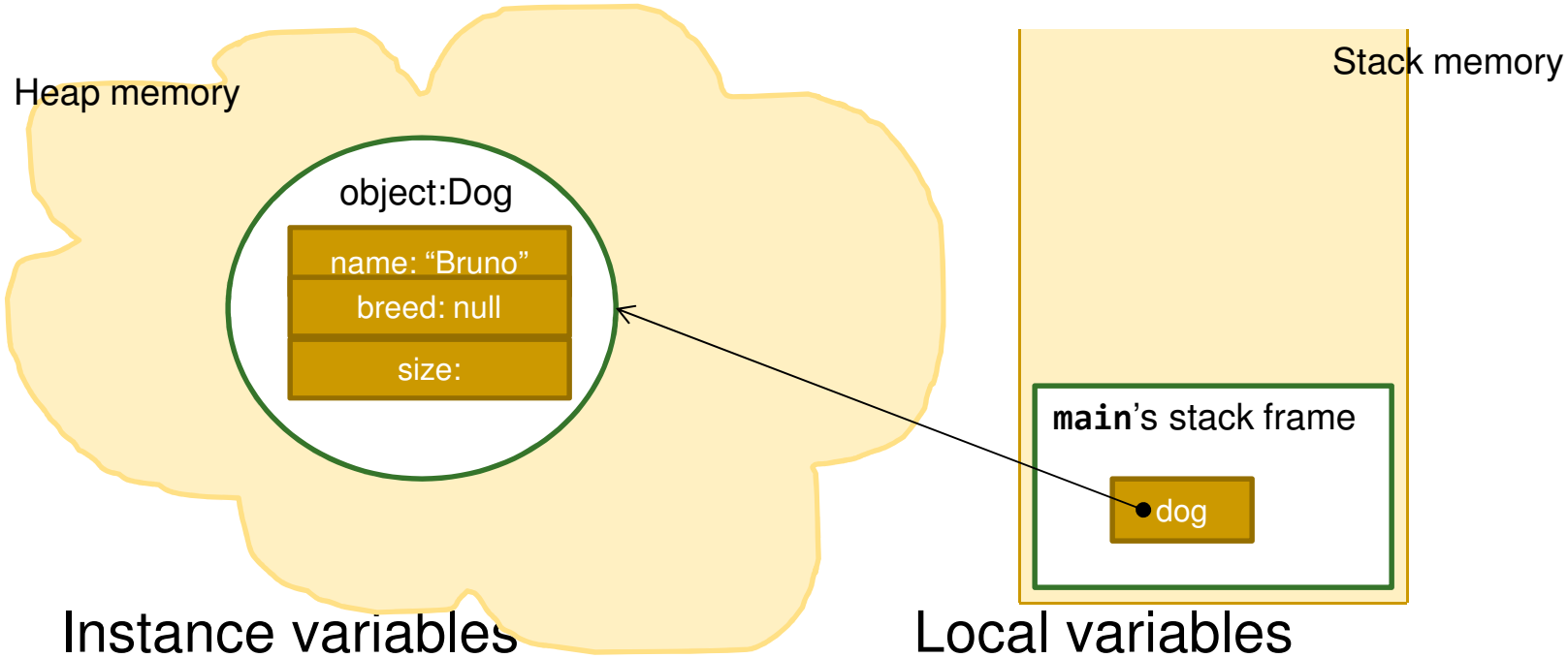
Instance variables vs. local variables




```
class Dog {
  int size;
  String breed;
  String name;
  ...
}
```

```
public class DogTestDrive {
  public static void main(String [
    Dog dog = new Dog();
    dog.name = "Bruno";
    dog.bark();
  }
}
```

Instance variables vs. local variables



Instance variables

- belong to an object
- located inside the object in the heap memory
- has the same lifetime as the object

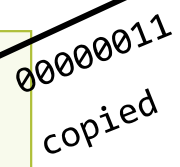
Local variables

- belong to a method
- located inside the method's frame in the stack memory
- has the same lifetime as the method call.

Parameter passing & return value

- Java allows only **pass-by-value**
  - That means pass-by-copy
  - Argument's content is copied to the parameter

```
Dog d = new Dog();
d.bark(3);
```

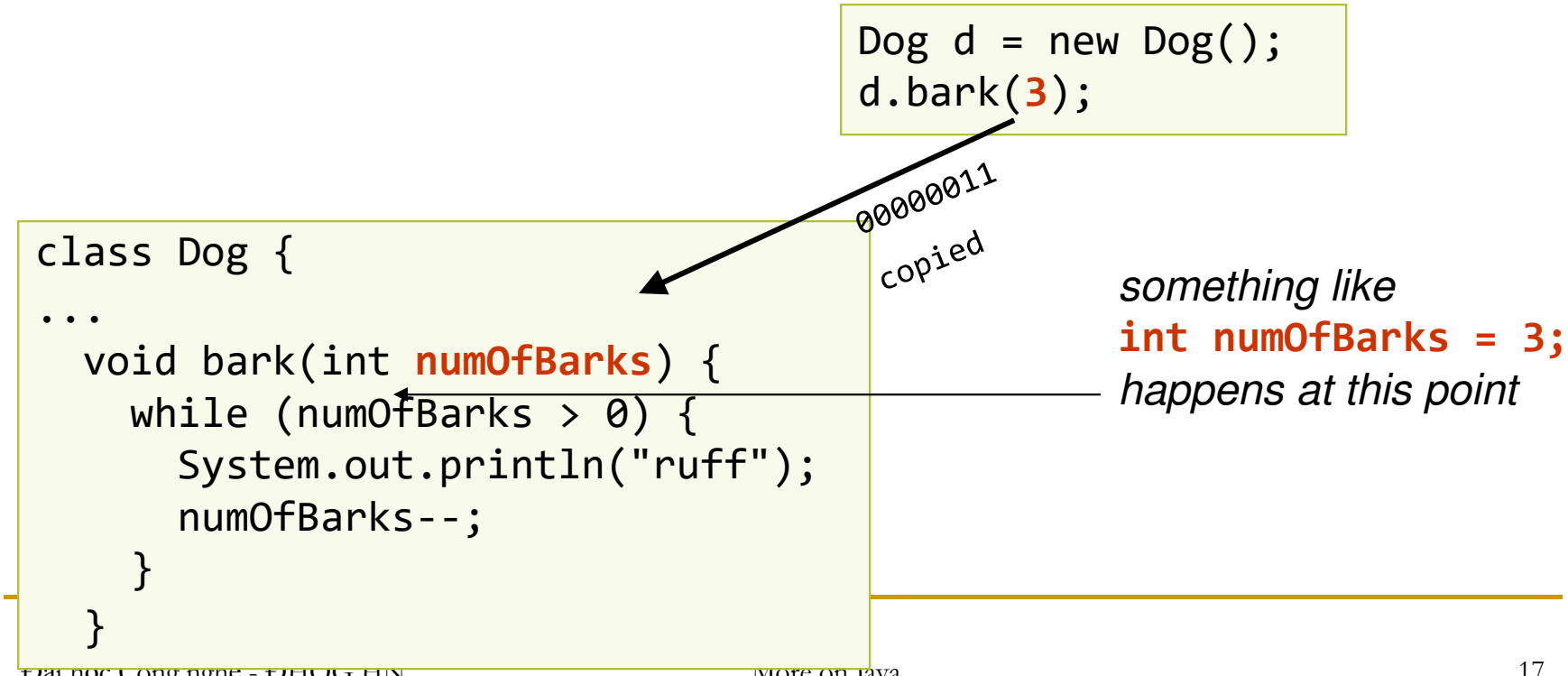



```
class Dog {
...
  void bark(int numOfBarks) {
    while (numOfBarks > 0) {
      System.out.println("ruff");
      numOfBarks--;
    }
  }
```

A method uses parameters.
A caller passed arguments

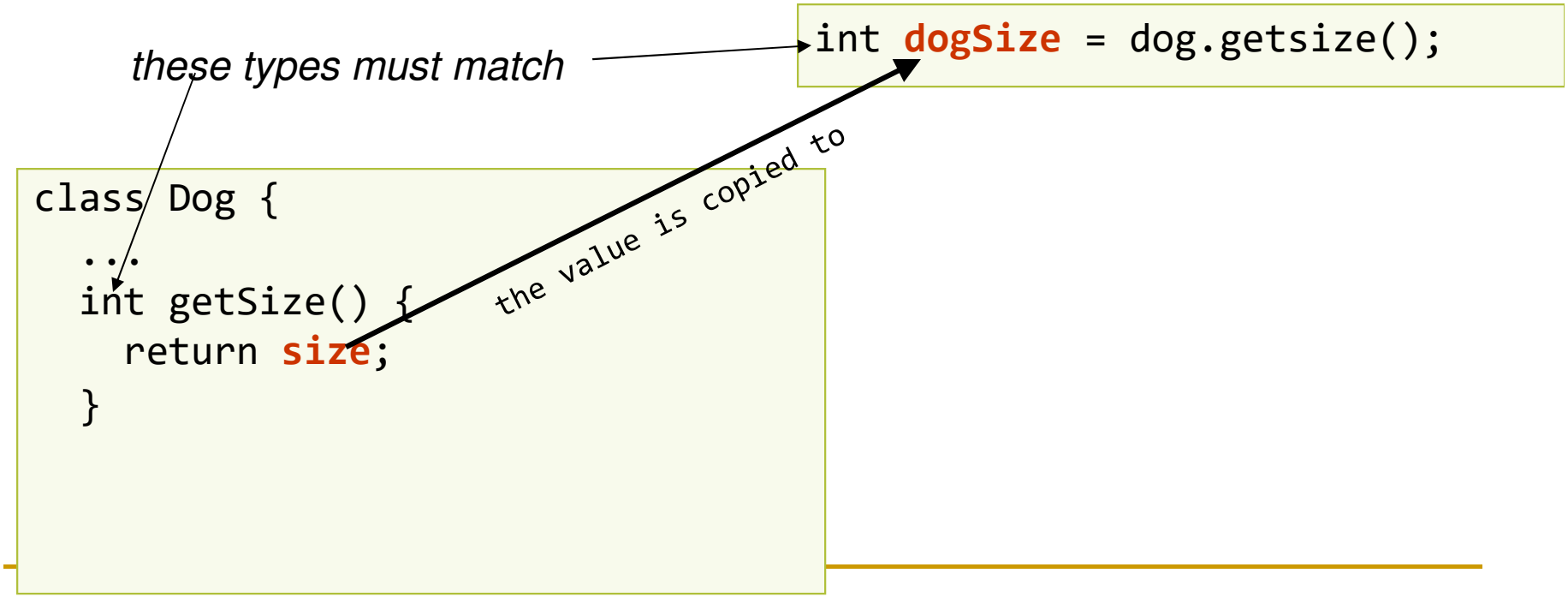
Parameter passing & return value

- A parameter is effectively a local variable that is initialized with the value of the corresponding argument.



Parameter passing & return value

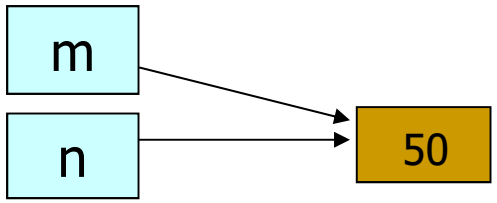
- The return value is copied to the stack, then to the variable that get assigned (dogSize in this example)



Parameter passing & return value

Two kinds of parameters:

- Primitive types
  - parameter's value is copied
  - parameters can be constants, e.g. 10, "abc"…
- Object references
  - the reference's value is copied, NOT the referred object.



Example

```
class Date {
    int year, month, day;
    public Date(int y, int m, int d) {
        year = y; month = m; day = d;
    }
    public void copyTo(Date d) {
        d.year = year;
        d.month = month;
        d.day = day;
    }
    public Date copy() {
        return new Date(day, month, year);
    }
...
}
```

y, **m**, **d** are of primitive data type. They'll take the values of the passed parameters.

d is a reference.
d will take the values of the passed parameter, which is an object location.

return a reference to the newly created Date object.
Again, it's a value, not the object

Example

```
...
int thisYear = 2010;
Date d1 = new Date(thisYear, 9, 26);
```

```
y = thisYear;
m = 9;
d = 26;
year = y;
month = m;
day = d;
```

```
class Date {
    int year, month, day;
    public Date(int y, int m, int d) {
        year = y; month = m; day = d;
    }
    public void copyTo(Date d) {
        d.year = year;
        d.month = month;
        d.day = day;
    }
    public Date copy() {
        return new Date(day, month, year);
    }
...
}
```

Example

```
...
Date d1 = new Date(thisYear, 9, 26);
Date d2 = new Date(2000, 1, 1);
d1.copyTo(d2);
```

```
class Date {
    int year, month, day;
    public Date(int y, int m, int d) {
        year = y; month = m; day = d;
    }
    public void copyTo(Date d) {
        d.year = year;
        d.month = month;
        d.day = day;
    }
    public Date copy() {
        return new Date(day, month, year);
    }
...
}
```

```
d = d2;
d.year = d1.year;
d.month = d1.month;
d.day = d1.day;
```

Example

```
...
Date d2 = new Date(2000, 1, 1);
Date d3 = d2.copy();
```

```
class Date {
    int year, month, day;
    public Date(int y, int m, int d) {
        year = y; month = m; day = d;
    }
    public void copyTo(Date d) {
        d.year = year;
        d.month = month;
        d.day = day;
    }
    public Date copy() {
        return new Date(year, month, day);
    }
...
}
```

```
Date temp =
  new Date(d2.year, d2.month, d2.day);
d3 = temp;
```

Method overloading

- Methods of the same class can have the *same name* but *different parameter lists*.

```
class Dog {
  ...
  void bark() {
    System.out.println("Ruff! Ruff!");
  }
  void bark(int numOfBarks) {
    while (numOfBarks > 0) {
      System.out.println("ruff");
      numOfBarks--;
    }
  }
}
```

```
Dog d = new Dog();

d.bark();

d.bark(3);
```

Do you still remember?

Instance variables/methods belong to an object.

Thus, when accessing them, you MUST specify **which object** they belong to.

dot notation (.) and ***the object reference***

```
public class DogTestDrive {
  public static void main(String [] args) {
    Dog d = new Dog();
    d.name = "Bruno";
    d.bark();
  }
}
```

access 'name' of ***the Dog***

call ***its*** *bark() method*

How about this case?

```
class Dog {
  int size;
  String breed;
  String name;

  void bark() {
    if (size > 14)
      System.out.println
    else
      System.out.println
  }
  void getBigger() {
    size += 5;
  }
}
```

Which object does **size** *belong to?*

the object that owns the current method – bark() or getBigger()

```
dog1.bark(); //this dog's size get compared

dog2.getBigger(); //this dog's size get increased
```

where is the object reference and dot notation?

The `this` reference

```
class Dog {
  int size;
  String breed;
  String name;

  void bark() {
    if (this.size > 14)
      System.out.println
    else
      System.out.println
  }
  void getBigger() {
    this.size += 5;
  }
}
```

this reference was omitted

```
dog1.bark(); //this dog's size get compared

dog2.getBigger(); //this dog's size get increased
```

The `this` reference

- **`this`** : the object reference referring to the current object – the owner of the current method
- usage of **`this`**:
  - explicit reference to object's attributes and methods
    - often omitted
  - parameter passing and return value
  - calling constructor from inside constructor

The `this` reference

```
class MyInteger {
  private int value;
  public boolean greaterThan (MyInteger other) {
    return (this.value > other.value);
  }
  public boolean lessThan (MyInteger other) {
           return (other.greaterThan(this));
  }
  public MyInteger increment() {
    value++;
    return this;
  }
}
```

```
MyInteger counter = new MyInteger();
counter.increment().increment(); // increased by 2
```

Input / output

- Details:
  - HFJ. Ch.14 / GT. Ch.12
- In this slide:
  - standard input / output stream
  - simple input / output
  - simple text file input / output

Standard I/O

- Three stream objects automatically created when a Java program begins executing:
  - **System.out** : standard output stream object
    - enables a program to output data to the console
  - **System.err** : standard error stream object
    - enables a program to output error messages to the console
  - **System.in** : standard input stream object
    - enables a program to input bytes from the keyboard

- Redirect at command line (input and output stream only):

```
C:\> type input.dat | java AJavaProgram > output.dat
```

Standard output and error streams

- System.out and System.err can be used directly
  - `System.out.println("Hello, world!");`
  - `System.err.println("Invalid day of month!");`
- Note: if you mix up these two streams in your programs, the output might not end up being displayed in the same order as the output instructions.

Standard input

- System.in
  - An InputStream object
  - must be wrapped before use
- Scanner: wrapper that supports input of primitive types and character strings
  - next(): get the next word separated by white spaces
  - nextInt(), nextDouble(),…: get the next data item
  - hasNext(), hasNextInt(), hasNextDouble(),…: check if there are data left to be read

Standard input. Example

```
// import the wrapper class
import java.util.Scanner;
...
// create Scanner to get input from keyboard
Scanner input = new Scanner( System.in );

// read a word
String s = sc.next());

// read an integer
int i = sc.nextInt();

// read a series of big intergers
while (sc.hasNextLong()) {
   long aLong = sc.nextLong();
}
```

Input from a text file. Example

Import required classes

To deal with errors such as file-not-found

Open and close the text file

```
import java.util.Scanner;
import java.io.FileInputStream;
import java.io.IOException;
...
public static void main(String args[]) {
   try {
      // create Scanner to get input from a file stream
      Scanner sc = new Scanner(new FileInputStream("test.dat"));

      String s = sc.next(); // read a word
      int i = sc.nextInt(); // read an integer
      while (sc.hasNextLong()) { // read a series of big intergers
         long aLong = sc.nextLong();
      }

      sc.close();

   } catch(IOException e) {
        e.printStackTrace();
   }
}
...
```

Write to a text file. Example

```
import java.io.PrintWriter;
import java.io.FileWriter;
import java.io.IOException;
...
public static void main(String args[]) {
   int i = 1; long l = 10;
   try {
        // create a printwriter to write output to a file stream
        PrintWriter out = new PrintWriter(new FileWriter("test.data"));

        // write to file
        out.println("Hello " + i + " " + l);

        out.close();
   } catch(IOException e) {
        e.printStackTrace();
   }
}
...
```

Command-line parameters

```
//CmdLineParas.java: read all command-line parameters
public class CmdLineParas {
   public static void main(String[] args)
   {
      //display the parameter list
      for (int i=0; i<args.length; i++)
         System.out.println(args[i]);
      }
   }
}
```

```
C:\>java CmdLineParas hello world
hello
world
```