Object-Oriented Programming

Objects and Object References

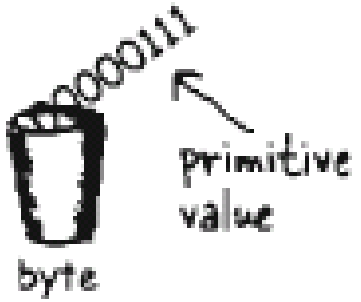
Contents

- Instance variables vs. local variables
- Primitive vs. reference types
- Object references, object equality
- Objects' and variables' lifetime
- Parameter passing and return values
- Method overloading
- this reference
- Simple input/output
- Packages

Variables and Types

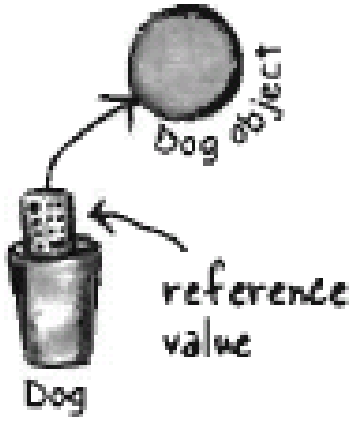
- Two kinds of variables: *primitive* and *object reference*
- primitive variables hold fundamental types of values: int, float, char,…

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



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

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



Primitive Data Types

- Three basic categories:
  - Numerical: byte, short, int, long, float, double
  - Logical: boolean (true/false)
  - Character: char

→ Primitive data are NOT objects

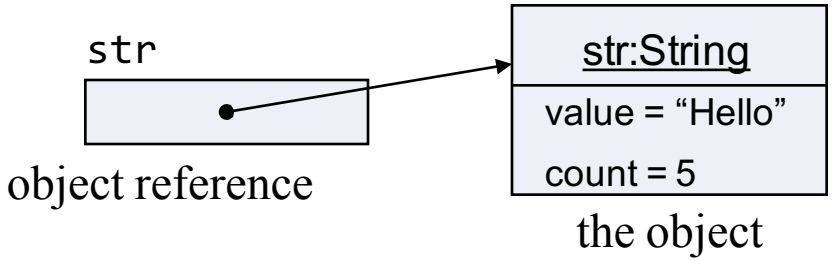
- wrapper type in order to treat primitive values as objects:
  - Integer, Float, Byte, Double, Character,…
    - Integer count = new Integer(0);
  - Provide utility functions: parseInt(), equals()…

Primitive Data Types

Primitive Type	Size	Minimum Value	Maximum Value	Wrapper Type
char	16-bit	Unicode 0	Unicode 216-1	Character
byte	8-bit	-128	+127	Byte
short	16-bit	-2^{15} (-32,768)	$+2^{15}-1$ (32,767)	Short
int	32-bit	-2^{31} (-2,147,483,648)	$+2^{31}-1$ (2,147,483,647)	Integer
long	64-bit	-2^{63} (-9,223,372,036,854,775,808)	$+2^{63}-1$ (9,223,372,036,854,775,807)	Long
float	32-bit	Approx range 1.4e-045 to 3.4e+038		Float
double	64-bit	Approx range 4.9e-324 to 1.8e+308		Double
boolean	1-bit	true or false		Boolean

Object References – Controlling Objects

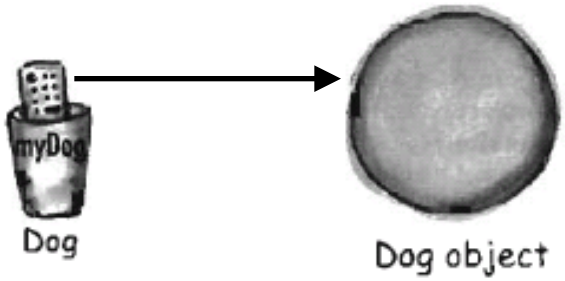
```
str = new String("Hello");
```

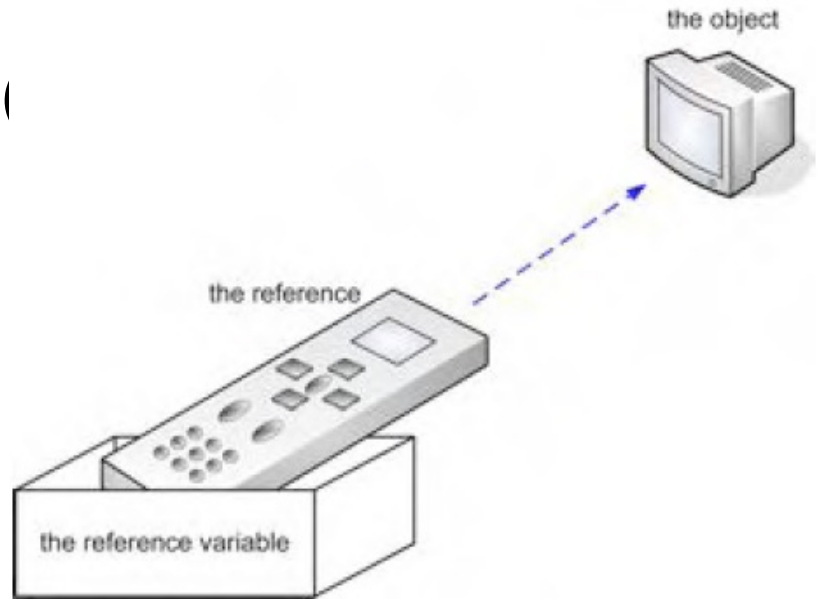


- There is actually no such thing as an *object variable*
- There‘re only ***object reference variables***
- An object reference variable represents a way to access an object, something like a pointer
- Think of an object reference as a remote control

Object References

```
Dog myDog = new Dog(
```



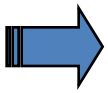


Remind: **References are not objects!**

Object Equality

- "==" and "!=" compares references (not objects) to see if they are referring 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.println(m1.equals(m2));
```

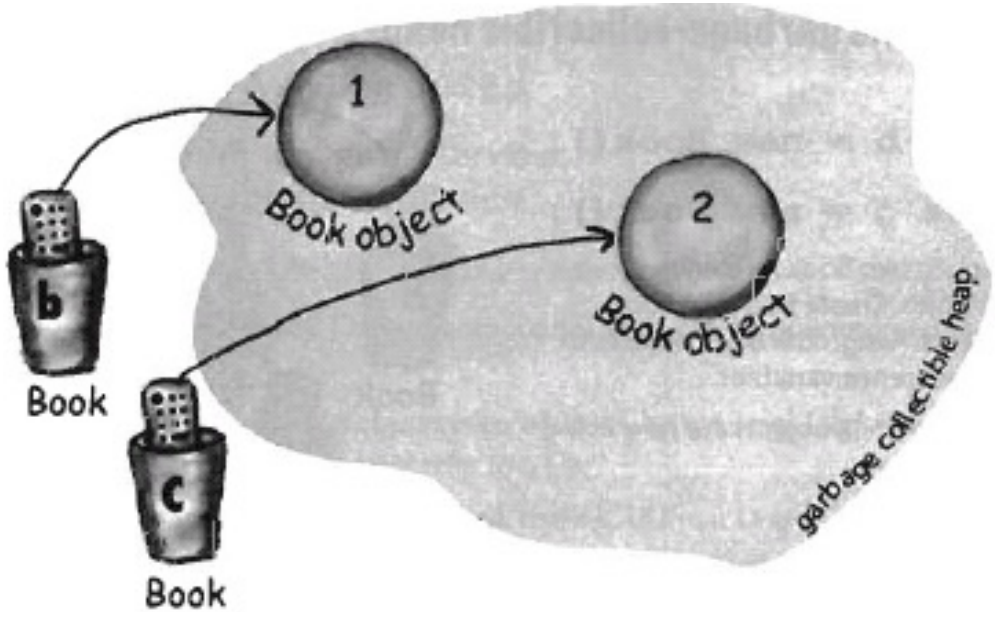
- User-created classes:
  - equals() must be defined, otherwise, it always returns *false*

```
class MyInteger {
    private int value;
    public boolean equals (Object other) {
       if (!(other instanceof MyInteger)) return false;
       return (value == ((MyInteger) other).value);
    }
}
```

Object's life on memory

- 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

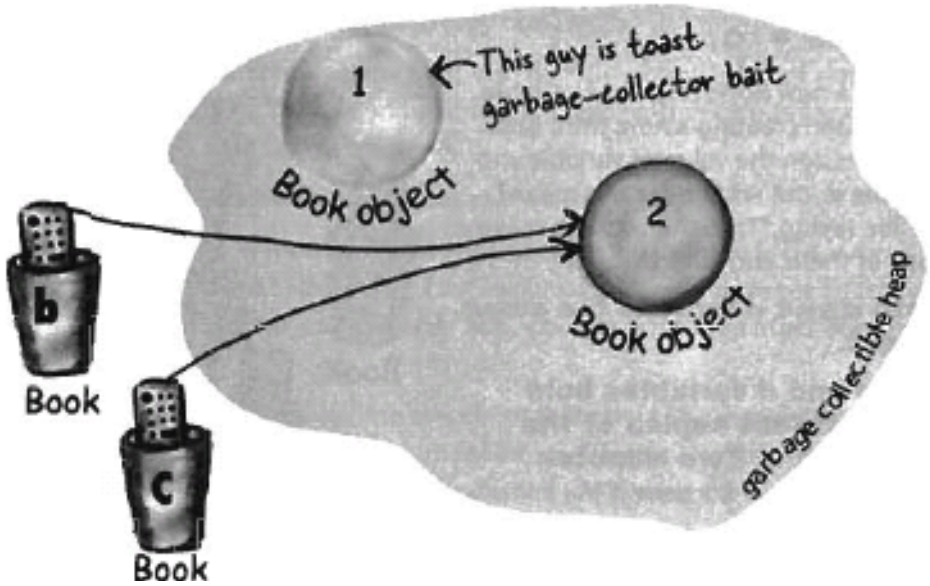
```
Book b = new Book();
Book c = new Book("Harry Potter");
```



Object's life on memory

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

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

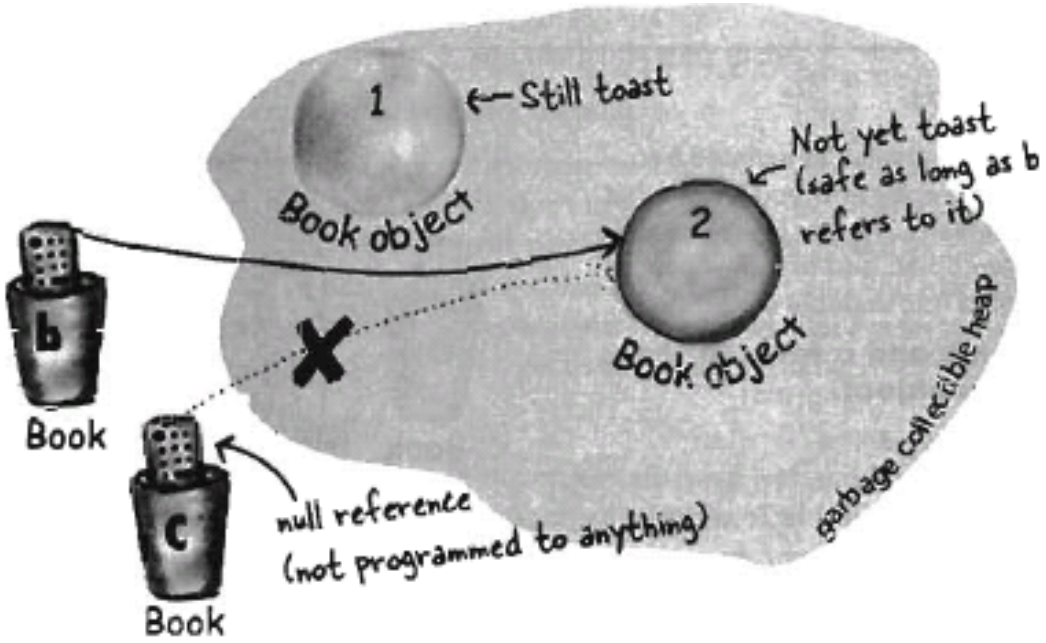


There is no way to reach Book object 1.

It is ready to be collected.

Object's life on memory

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

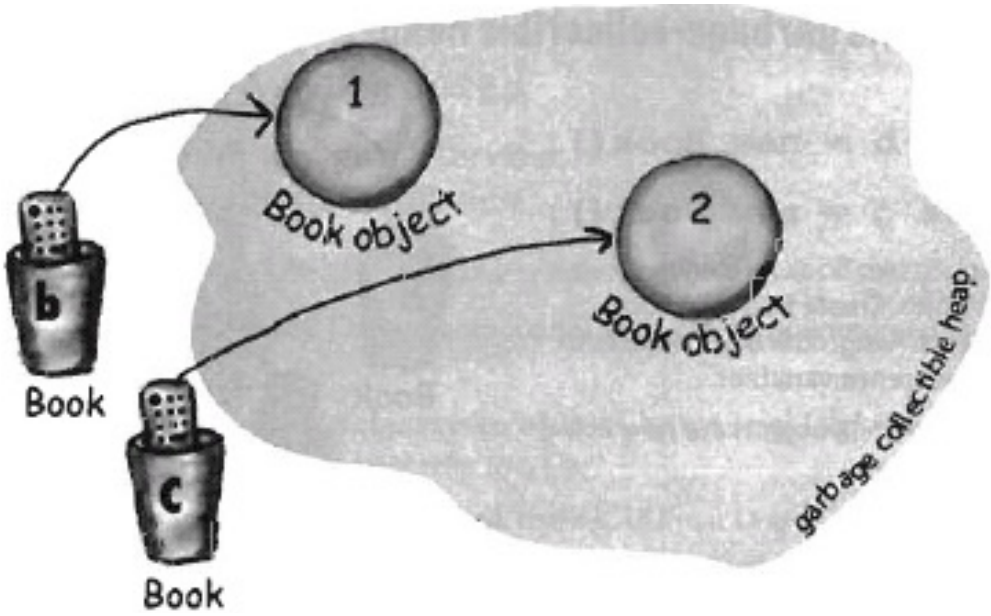


Book object 1 is waiting to be de-allocated.

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

Object's life on memory

- In Java, un-used objects are automatically freed by Java Virtual Machine (JVM), not manually by programmers



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

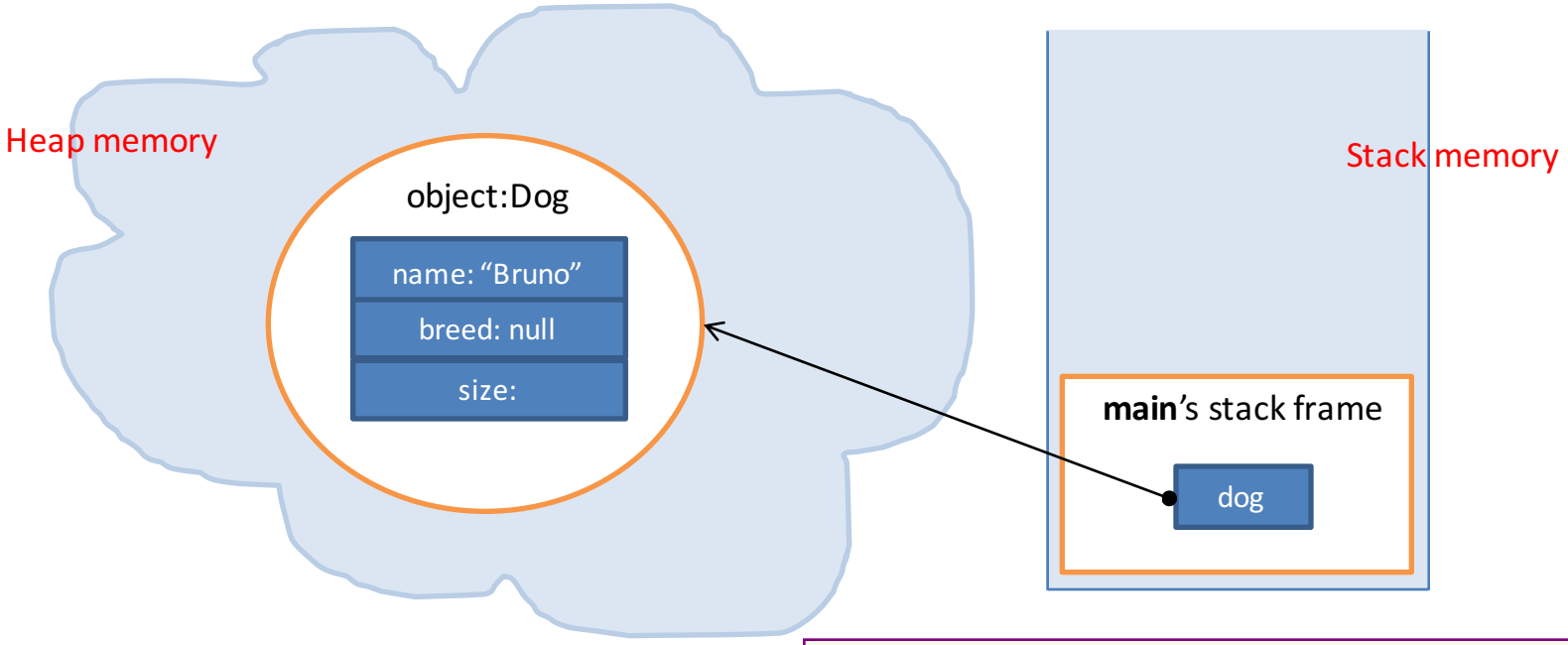
Local variables

- 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
   [] args) {
    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
   [] args) {
    Dog dog = new Dog();
    dog.name = "Bruno";
    dog.bark();
  }
}
```

Parameter Passing & Return Value

- Parameter: used in method definition or declaration
- Argument: used in method call

A parameter

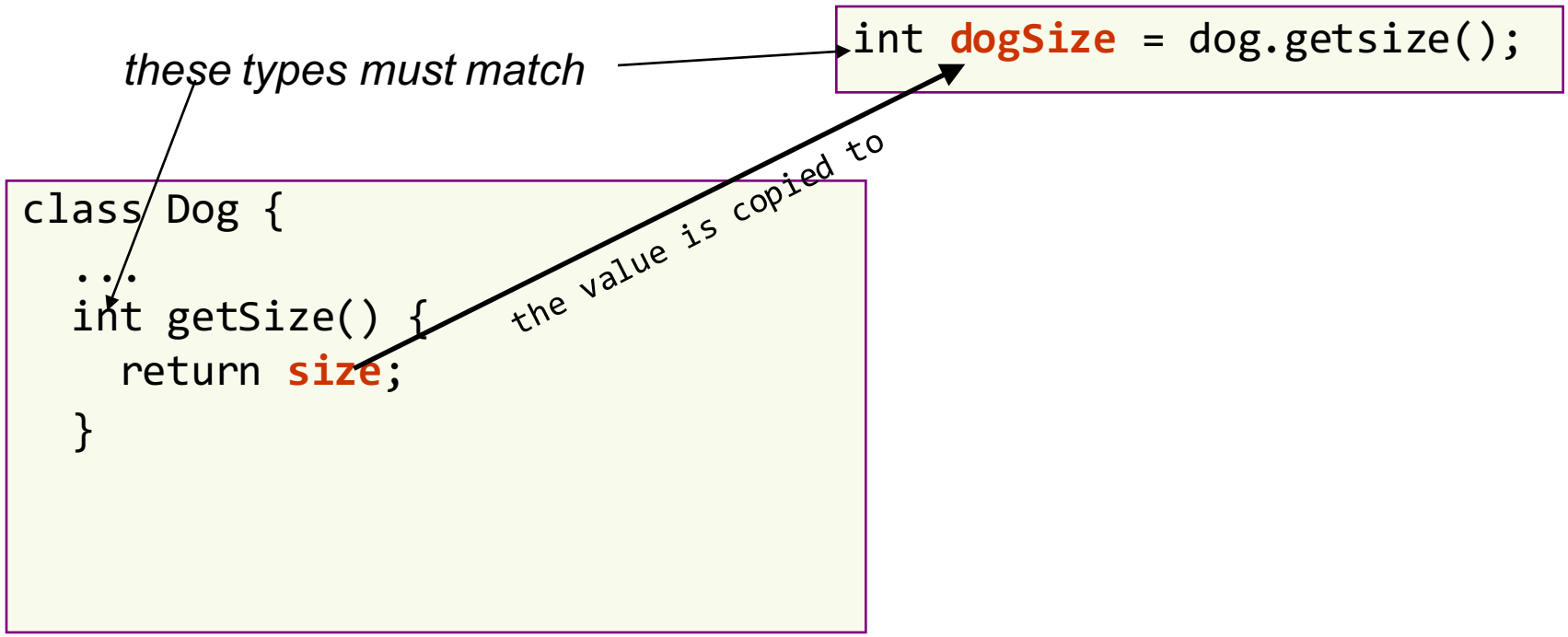
```
class Dog {
...
  void bark(int numOfBarks) {
    while (numOfBarks > 0) {
      System.out.println("ruff");
      numOfBarks--;
    }
  }
```

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

An argument

Parameter Passing & Return Value

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

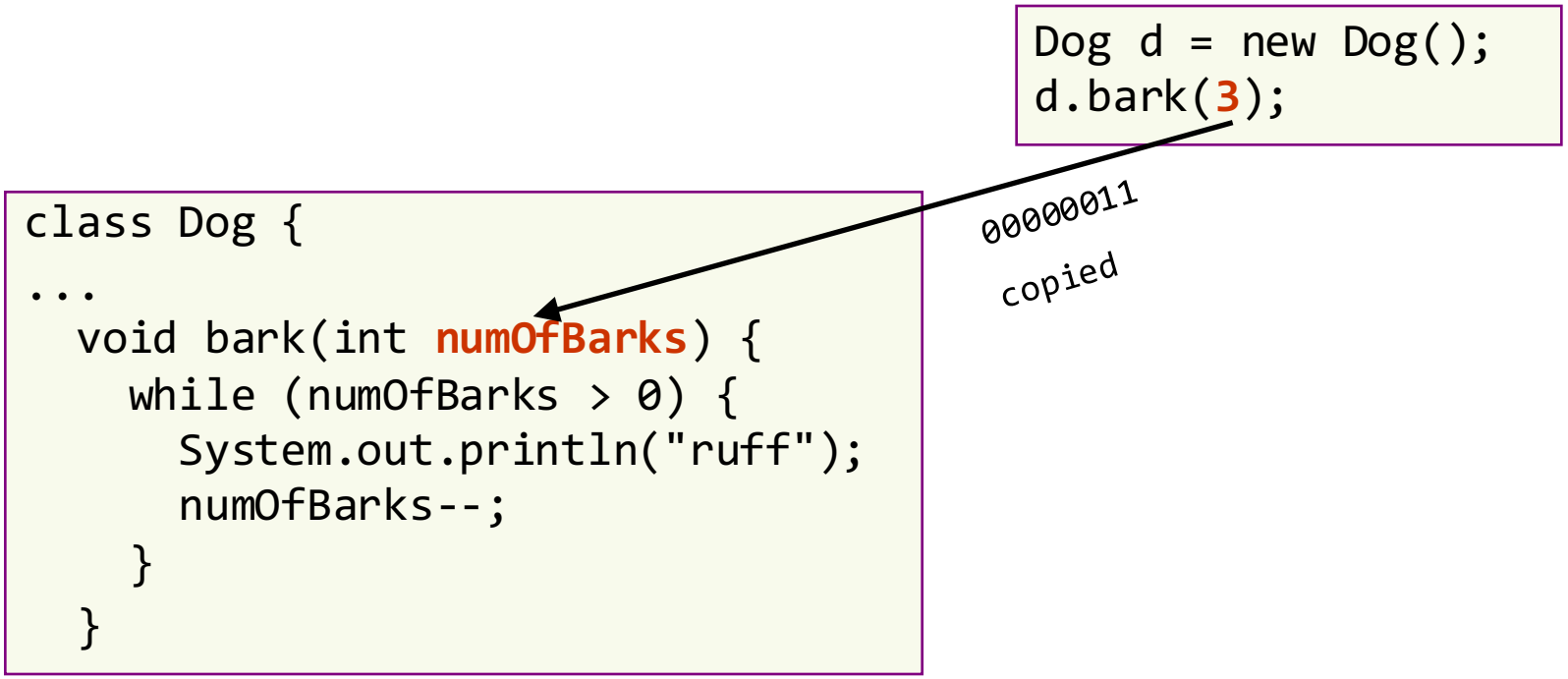


Parameter Passing & Return Value

- 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

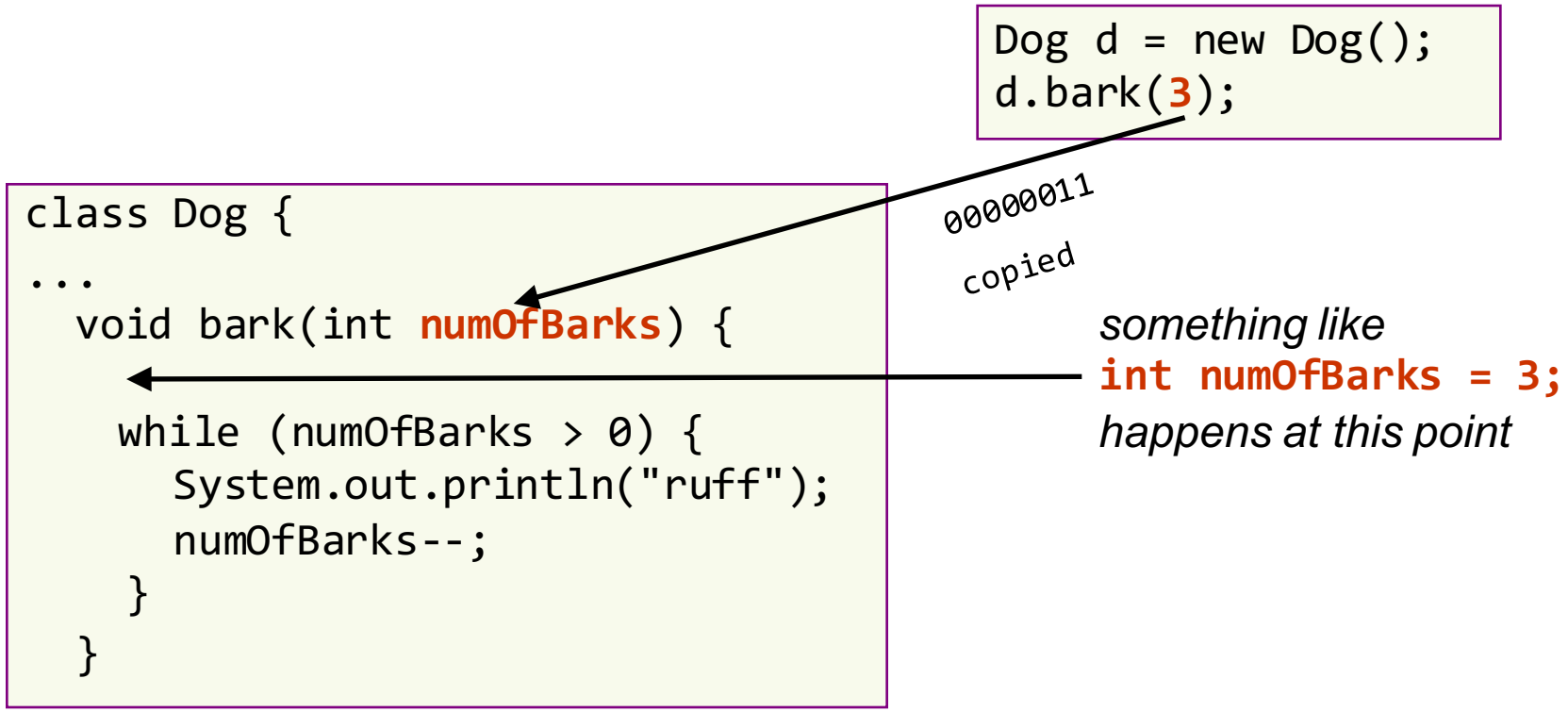
Parameter Passing of Primitive Types

- pass-by-copy:
  - Argument's content is copied to the parameter



Parameter Passing of Primitive Types

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



Parameter Passing of Object References

- Object reference's value is copied, NOT the referred object

```
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 types. They'll take the values of the passed parameter

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

Parameter Passing of Object References

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

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

Parameter Passing of Object References

```
...
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;
```

Parameter Passing of Object References

```
...
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);
```

Remind

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("Ruff! Ruff!");
    else
      System.out.println("Yip! Yip!");
  }
  void getBigger() {
    size += 5;
  }
}
```

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

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

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("Ruff! Ruff!");
    else
      System.out.println("Yip! Yip!");
  }
  void getBigger() {
    this.size += 5;
  }
}
```

this reference was omitted in the previous slide

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 another 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
```

The "this" reference

```
class MyInteger {
  public int value;

  public MyInteger(int initialValue) {
    value = initialValue;
  }

  public MyInteger() {
    this(0);
  }

  public MyInteger(MyInteger other) {
    this(other.value);
  }
}
```

Calls to MyInteger(int)

Input / Output

- In Java, input and output are often performed on data streams
- A stream is a sequence of data. There are two kinds of streams:
  - InputStream: to read data from a source
  - OutputStream: to write data to a destination
- Most I/O classes are supported in java.io package

Standard I/O

- Three stream objects are 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 data from the keyboard

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!");
```

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 sc = new Scanner(System.in);

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

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

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

Input from a text file: Ex

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.txt"));

      String s = sc.next(); // read a word
      int i = sc.nextInt(); // read an integer
      while (sc.hasNextLong()) { // read a series of big integers
         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.txt"));

        // 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]);
      }
   }
}
```

Package

- A package is a grouping of related types (e.g. classes, interfaces, etc.) to protect access or manage namespace
- Two popular packages:
  - java.lang: bundles the fundamental classes (System, String, Math, etc.)
  - java.io: bundles classes for input/output functions (FileInputStream, PrintWriter, FileWriter, etc.)

Create a Package

- Task: create a package named "messagePkg" contains the two following classes:

```
public class HelloMessage {
    public void sayHello() {
        System.out.println("Hello Everyone!");
    }
}
```

```
public class WelcomeMessage {
    public void sayWelcome() {
        System.out.println("Welcome ICTBI6 Class!");
    }
}
```

Create a Package

- Step 1: declare the package which the class belongs to:

```
package messagePkg;

public class HelloMessage {
    public void sayHello() {
        System.out.println("Hello Everyone!");
    }
}
```

package declaration with package name. The rest of the file belongs to the same package

```
package messagePkg;

public class WelcomeMessage {
    public void sayWelcome() {
        System.out.println("Welcome ICTBI6 Class!");
    }
}
```

Create a Package

- Step 1: declare the package which the class belongs to:

```
package messagePkg;

public class HelloMessage {
    public void sayHello() {
        System.out.println("Hello Everyone!");
    }
}
```

```
package messagePkg;

public class WelcomeMessage {
   public void sayWelcome(){
        System.out.println("Welcome ICTBI6 Class!");
   }
}
```

Declared as **public** so that
they can be used outside package messagePkg

Create a Package

- Step 2: Compile the classes of the same package:

```
javac -d <destination_folder> file_name.java
```

- Example:

```
javac -d . HelloMessage.java
javac -d . WelcomeMessage.java
or:
 javac -d . HelloMessage.java WelcomeMessage.java
```

Try it by yourself to see how it works!

Use a Package

- Two ways:

1. Use the **import** statement to make the name(s) in the package available, once for all

```
import messagePkg.HelloMessage;

public class Hello {
    public static void main(String[] args) {
        HelloMessage msg = new HelloMessage ();
        msg.sayHello();
    }
}
```

2. Give the fully qualified name at every call

```
public class Hello {
  public static void main(String[] args) {
    messagePkg.HelloMessage msg = new messagePkg.HelloMessage();
    msg.sayHello();
  }
}
```

Use a Package

- Compile

```
javac Hello.java
```

- Run

```
java Hello
```

Try it by yourself to see how it works!