

Garbage Collection



covering

- ** Objects and Variables: Heap *versus* Stack
- ** (Overloaded) Constructors, `this()`, `super()`
- ** Life of an Object // Scope of a Variable
- ** Garbage Collector // `null` References



Chapters 4+5, (sections 4.3, 4.6, 5.1) – “Big Java” book

Chapter 9 – “Head First Java” book

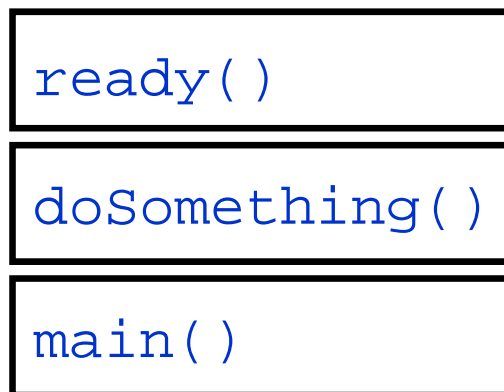
Chapters 5, 8, 10 (sections 5.9, 8.4-8.5, 10.3-10.4) – “Introduction to Java Programming” book

Chapter 6 – “Java in a Nutshell” book

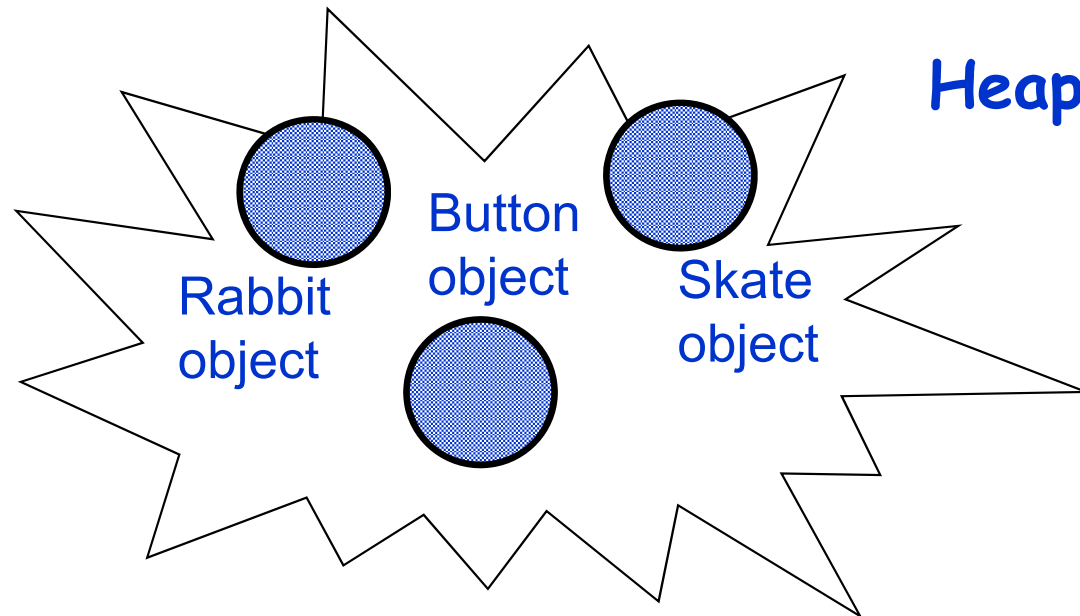
Heap versus Stack

- **Running a Java program:**
 - Memory is obtained by the JVM, from the OS being used.
- **Two main areas of memory in Java:**
 - (Garbage-collectible) **Heap** (where **objects** live);
 - **Stack** (where **local variables** and **methods**, when called, live).

Stack



Heap



Local Variables & Instance Variables

- Main types of variables we care about:
 - **Local** (also known as **stack**) variables
 - Variables declared in a method and method parameters.
 - Temporary variables, alive only when the method they belong to is on the *Stack*.
 - **Instance** variables
 - Variables declared in a class (**not** inside of a method).
 - Live inside the object they belong to.



So where do **instance variables** live?

Example: Local *versus* Instance Variables

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

instance variables; every **Dog**
has a “size” and “name”

```
private int maxDistanceRun(int timeRun) {  
    int maxSpeed = 10;  
    int maxDistance = maxSpeed * timeRun;  
    return maxDistance;  
}
```

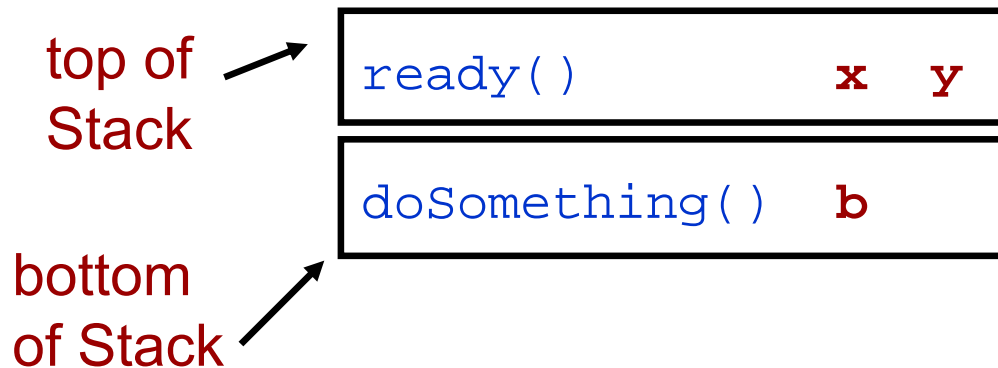
local variables



What is the difference between
instance and **class** variables?

Methods and the Stack

- Method goes on top of the *Stack* when it is called and stays in the *Stack* until it's done (closing curly brace).
- **Stack frame:**
 - What actually is pushed onto the *Stack*.
 - Contains the **state of the method** (which line of code is executing and values of all local variables).
- Method at top of the *Stack* is always the method being executed.
- **Example:**



```
public void doSomething() {  
    boolean b = true;  
    ready(10);  
}  
  
public void ready(int x) {  
    int y = x * 24;  
    // more code here  
}
```

Object References & Where Variables Live

- **Object reference** (*aka non-primitive*) variables:
 - Hold a reference to an object, not the actual object.
 - A local variable that is a reference to an object goes on the *Stack* (the object it refers to still goes on the *Heap*).
- **Where variables** (primitive and non-primitive types) **live**:
 - **Local** variables (on the *Stack*)
 - **Instance** variables (on the *Heap*)

Why learn about the *Stack* and *Heap*?

- **Required in order to understand,**
 - variable scope;
 - issues with creating objects;
 - memory management (*);
 - threads (**);
 - exception handling (*).

(*) Later in the course!
(**) Covered in advanced course!
- **Heap and Stack:**
 - Not necessary to know how they're implemented on a given JVM or platform!

Initialising Object State

- **Using a constructor to initialise object state** (i.e. instance variables):
 - Most common use of constructors.
 - **Constructors**: the best place to put initialisation code.
 - Programmers should always **write a “no arguments” constructor** (to **build an object with default values**): makes things easier for the program’s users.

– **Example:**

```
public class Dog {  
    private String name;  
    public Dog() {  
        System.out.println("Woof, Woof!");  
    }  
    public void setName(String newName) {  
        this.name = newName;  
    }  
}  
public class UseADog {  
    public static void main (String[] args) {  
        Dog myDog = new Dog();  
        myDog.setName("Rover");  
    }  
}
```

instance variable

constructor

mutator/setter method



Is this OK?

Example: Two Constructors for an Object

```
public class Dog {  
    private int weight;  
    public Dog() {  
        // Use a default weight (in Kg).  
        this.weight = 30;  
    }  
    public Dog(int dogWeight) {  
        // Use the dogWeight parameter.  
        this.weight = dogWeight;  
    }  
}
```



There is a *special name* for this type of constructors.

- Making a **Dog** in two different ways:

```
Dog myDog = new Dog(15);
```

```
Dog anotherDog = new Dog();
```

if you **know** the dog's weight

if you **don't know**
the dog's weight

Overloaded Constructors

- When you have **more than one constructor in a class**.
- Must have **different argument lists** (it's the variable type and order that matters);
- There may be cases where a no-arguments constructor makes no sense (e.g. making a **Color** object).
- **Example:**

```
public class Account {  
    public Account() {}  
    public Account(int balance) {}  
    public Account(boolean giveOverdraft) {}  
    public Account(int balance, boolean giveOverdraft) {}  
    public Account(boolean junior, int balance) {}  
}
```

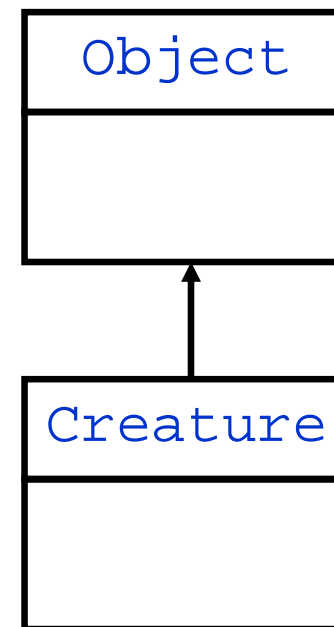
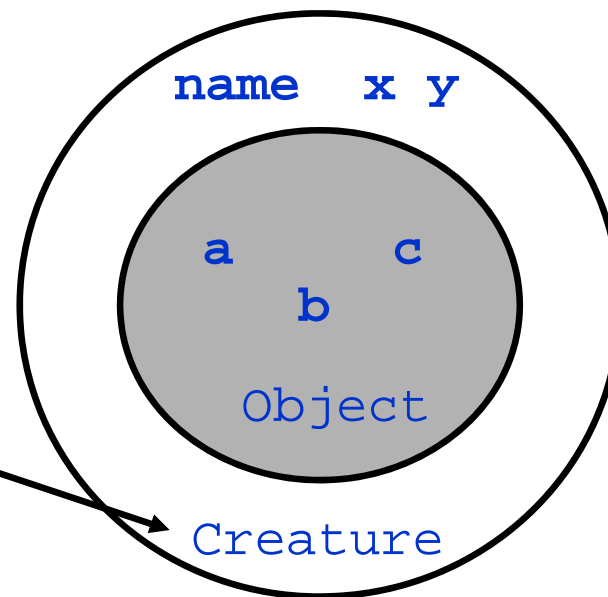
**no-arguments constructor, when
you just want to create an Account**

Superclasses, Inheritance & Constructors

- **How they relate:**

- Every **object** holds both its own declared instance variables and everything from its superclasses.
- When an object is created, that object will have “layers” of itself representing each superclass.
- **Remember:** Every class in Java extends class **Object** (it's the mother of all classes!).
- **Example:**

**a Creature object
on the Heap**



As a consequence, we have Constructor Chaining

- **Superclass constructors** (or constructor chaining):
 - When a **new object** is created, all the constructors in its inheritance tree must be run.
 - Saying **new** triggers a ‘chain reaction’ of constructors being run implicitly.
 - An **object is only completely formed** when all the superclass parts of itself are formed.

Example (to demonstrate): Constructor Chaining

```
public class BigCat {  
    public BigCat() {  
        System.out.println("Making a BigCat");  
    }  
}  
  
public class Tiger extends BigCat {  
    public Tiger() {  
        System.out.println("Making a Tiger");  
    }  
}  
  
public class TestTiger {  
    public static void main(String[] args) {  
        System.out.println("Starting ...");  
        Tiger myTiger = new Tiger();  
    }  
}
```

Output is ...

```
% java TestTiger  
Starting ...  
Making a BigCat  
Making a Tiger
```

Calling and Making Constructors

- Calling a superclass constructor:
 - using **super()** calls the super constructor;
 - using **super()** in your constructor **puts the superclass on the top of the stack.**



What does this trigger?

- **Example:**

```
public class Tiger extends BigCat {  
    private int size;  
    public Tiger(int newSize) {  
        super(); ←  
        size = newSize;  
        System.out.println("Making a Tiger");  
    }  
}
```

It is usually only used when we don't want to call the parent's no-args constructor (which means that we usually use **super()** with arguments).

- **Compiler and making constructors:**

- if no constructor is provided, the compiler adds one that looks like:

```
public ClassName() { super(); }
```

- if you provide a constructor but do not add a call to **super()**, the compiler puts such a call in each of your overloaded constructors.

Using `super()` (1/2)

- The **call to `super()`** must be the first statement in a constructor.

```
public Dog() {  
    super();  
}
```



Are all these constructors OK?

```
public Dog(int dogWeight) {  
    weight = dogWeight;  
    super();  
}
```

```
public Dog() {}
```

```
public Dog(int dogWeight) {  
    weight = dogWeight;  
}
```

Using super () (2/2)

- You can pass arguments into a call to **super ()**.

```
public class Creature {  
    private String name;  
    public String getName() {  
        return this.name;  
    }  
    public Creature(String aName) {  
        this.name = aName;  
    }  
}  
public class Rabbit extends Creature {  
    public Rabbit(String name) {  
        super(name);  
    }  
}
```

Output is ...

```
% java TestRabbit  
Bunny
```



What does this statement do here?

```
public class TestRabbit {  
    public static void main(String[] args) {  
        Rabbit aRabbit = new Rabbit("Bunny");  
        System.out.println(aRabbit.getName());  
    }  
}
```


Overloaded constructors, `super()` and `this()`

- To call a constructor from another overloaded one in the same class, use **`this()`** – must be the first statement in the constructor.
- A constructor can have a call to either **`super()`** or **`this()`** but not to both!

```
public class Tiger extends BigCat {  
    private int speed;  
  
    public Tiger() {  
        this(5);  
    }  
    public Tiger(int newSpeed) {  
        super("Tiger");  
        this.speed = newSpeed;  
        // maybe more initialisation  
    }  
}
```

cont.

```
public Tiger(String newName) {  
    this(5);  
    super(newName);  
}
```

```
% javac Tiger.java  
Tiger.java:14: call to  
super must be first  
statement in constructor  
    super();  
      ^
```

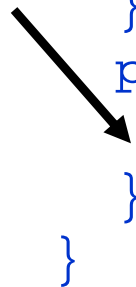
Output is ... 

Life of Objects and Variables (1/2)

- Life of an object: **depends** only on the life of reference variables referring to it.
 - Object is **alive** (or **dead**) if its reference is **alive** (or **dead**).
- Variable lifetime:
 - **same** for **primitive** and **reference** variables;
 - **different** for **local** and **instance** variables.

```
public class ExampleGoneWrong {  
    public void method1() {  
        int x = 10;  
        method2();  
    }  
    public void method2() {  
        x = 20;  
    }  
}
```

local variable **x** is
out of scope here



Is **x alive here?**



... and things for you to try out!

Life of Objects and Variables (2/2)

- Life duration:
 - **local variables**: live only within the method that declared it (also referred to as *being in scope*);
 - **instance variables**: live for as long as object they belong to lives.

```
public class Dog {  
    private String name;  
    // name can be used throughout the class.  
    public void setName(String dogName) {  
        this.name = dogName;  
        // dogName disappears at the end of the method.  
    }  
    public void sleep() {  
        int x = 10;  
    }  
}
```

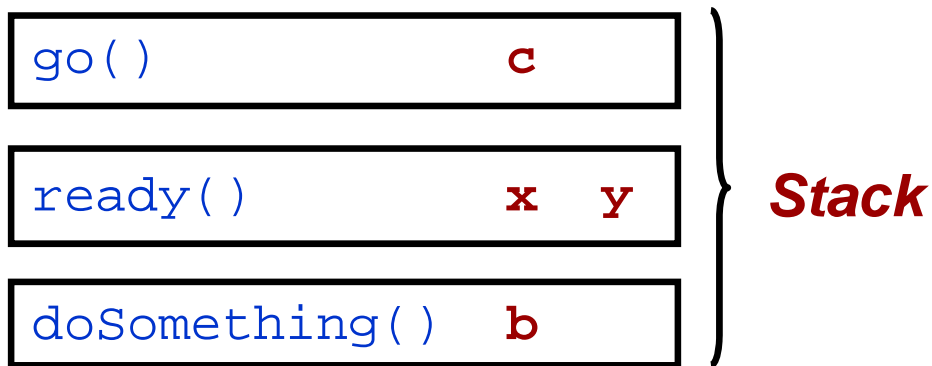


What about the *scope*
of variable **x**?

Local Variables: Life *versus* Scope

- Local Variable:

- Is **alive** as long as its *Stack* frame is on the *Stack*, i.e. until the method it belongs to completes.
- Is **in scope** only within the method where it was declared.
- Is **alive** (and maintains its state) **but is not in scope** when its own method calls another.
- A (reference) **variable can only be used when it is in scope**.



All 3 methods not
always on the **Stack**!

```
public void doSomething() {  
    boolean b = true;  
    ready(10);  
}  
public void ready(int x) {  
    int y = x * 24;  
    go();  
}  
public void go() {  
    char c = 'T';  
}
```

Practice Exercise 1

- Show the contents of the stack before `max()` is invoked, just entering `max()`, just before `max()` is exited, and after `max()` is exited.



What is the output of the program?

```
public class Test {  
    public static void main(String[] args) {  
        Test myTest = new Test();  
        int max = 0;  
        myTest.max(1, 2, max);  
        System.out.println(max);  
    }  
    public void max(int value1, int value2, int max) {  
        if (value1 > value2) { max = value1; }  
        else { max = value2; }  
    }  
}
```

Answer: Practice Exercise 1 *[to be completed in class ...]*

Space required for
the **main()** method

args: null
max: 0

just before
invoking **max()**

just entering
max()


just before exiting
from **max()**

just after
exiting **max()**

Life of an Object (again) & GC (1/2)

- Objects in Java are *dynamically* allocated and created *on demand*:
 - memory space for an object is allocated at *runtime*, not at *compile time*;
 - the **new** statement causes the memory for an object to be allocated (similar to the **C malloc()** function).
- As a consequence, the *memory taken up by a program will grow (and shrink)* as the program executes.
- **Example:**

```
for (long i = 0; i < 100000000; i++) {  
    Date d = new Date();  
}
```



What happens here?
- Once a **reference to an object is lost**, it can never be regained!
 - The object is still taking up memory, but cannot be used by the program.
- **Memory Leak:**
 - In **C**, this is a classic reason for program failure.
 - In **Java**, we don't need to worry about this: it handles *garbage collection* for us.

Life of an Object (again) & GC (2/2)

- An **object is alive as long as there are live references to it**.
 - If **an object has only one reference to it** and the *Stack* frame holding it gets popped off the *Stack*, then the object is now abandoned in the *Heap*.
 - In this case, the object becomes eligible for **Garbage Collection (GC)**.
- **Objects eligible for GC:**
 - **Programmers** don't need to reclaim their used memory, but they **need to make sure objects are abandoned**, when appropriate.
 - If a **program gets low on memory**, the **GC will destroy as many eligible objects as possible**, to avoid the program running out of RAM.



This is not full proof!

Making an Object Eligible for GC

- Ways to get rid of an object's references:
 - (1) The **reference goes out of scope**, permanently.
 - (2) The **reference is assigned to another object**.
 - (3) The **reference is explicitly set to null**.

```
public void doSomething() {  
    Car myCar = new Car(); (1)  
}
```


```
Car myCar = new Car();  
myCar = new Car(); (2)
```

```
Car myCar = new Car();  
myCar = null; (3)
```

- Note:** The opposite of a Java **constructor** is a **finalizer**; it can sometimes be used for the *cleanup* of an object.

null References

- **Setting a reference variable to `null`** (it means *no object*).

- 
- If you use the dot operator on a null reference, you will get a **`NullPointerException`** error at runtime.
 - Unless initialised/assigned to, instance reference variables have a default value of **`null`**.

```
public class Student {  
    String name;  
    int age;  
    boolean isMScStudent;  
    char gender;  
}
```

name not initialised so has
default value of **`null`**

default values are: **`0`**; **`false`**; **`'\u0000'`**

```
public static void main(String[] args) {  
    Student aStudent = null;  
    System.out.println("Student name? " + aStudent.name);  
}  
}
```

this will cause a **`NullPointerException`**
runtime error

Practice Exercise 2

- What is wrong with the code below?

```
public class Foo {  
    public void method1() {  
        Circle c = null;  
        System.out.println("What is radius " +  
                               c.getRadius());  
        c = new Circle();  
    }  
}
```

Assumption: **Circle** is a class we have defined somewhere else.



... and things for you to try out!