
ZIPCODE
<WILMINGTON>

# INITIALIZATION AND CLEANUP

# IMPORTANT TERMS

- Constructor: A special function that creates an instance of a class
- Overloading: Creating multiple methods with the same name but different argument lists.
- Declaration: Specifying the name and type of a variable
- Initialization: The first value assignment a variable or reference receives

# EXAMPLES: CONSTRUCTOR

```
class Foo{

// Constructor for Foo
  public Foo(){ ... }
}
```

# EXAMPLES: OVERLOADING

```
public static int sum(int x, int y){ return x + y;}

public static int sum(int[] nums){
  int total = 0;
  for(int i : nums){ total += i; }
  return total;
}
```

# EXAMPLES: DECLARATION

```
int x;
String s;
Document independence;
```

# EXAMPLES: INITIALIZATION

```
int pi = 3; // ಠ_ಠ
Repo myRepo = new Repo();
double realPi = pi + 0.14159265358979324;
```

# CONSTRUCTORS

# CONSTRUCTORS

- Constructors are special methods that perform set-up tasks for an instance of a class.
- Constructors are only called once in the life cycle of an instance.
- Constructors are named after the object class they belong to.

# CONSTRUCTORS ARE METHODS

- Constructors have parameters and access just like any other method
- Constructors have access to instance variables of an object
- Constructors have no return value or type

# CONSTRUCTORS ARE SPECIAL

Constructors have no return value (different from void return).
Constructors are called without an instance (but aren't static).
Cannot be called as member methods of an initialized instance.

# NO RETURN VALUE

```
public Kung(){...}
public Kung foo(){...}
public void bar(){...}
```

# CALLED WITHOUT INSTANCE

Constructors are not static, but do not require an instance to be called.

```
String str = new String("");
//     NOT:  str.String("");
```

# CANNOT BE CALLED AS MEMBER METHODS

```
Manchu foo = new Manchu();
foo.Manchu();
/***
* Foo.java:20: error: cannot find symbol
* foo.Manchu();
*     ^
*   symbol:   method Manchu()
*   location: variable foo of type Manchu
***/
```

# OVERLOADING METHODS

# OVERLOADING VS OVERRIDING

**Overloading -** Creating alternate versions of the same method
**Overriding -** replacing the existing method with a new one

# DISTINGUISHING OVERLOADED METHODS

Overloaded methods have distinct argument lists (number and type).
Three overloaded methods:

```
public int foo(int x) {return x;}
public boolean foo(boolean y) {return !y;}
public boolean foo(char c, int x) {return c == 's' || x > 5; }
```

# OVERLOADING RETURN VALUES

Not allowed in Java. Ambiguous example:

```
public int foo(){ System.out.println("int"); return 0; }
public void foo(){ System.out.println("void"); return; }
public static void bar(){ foo(); } //which foo?
```

# DEFAULT CONSTRUCTORS

# AUTOMATICALLY CREATED

If no constructor is defined, java automatically creates a no-arg constructor (aka default constructor)

```
class Feefie{}
public class Bar{
  private Feefie fofum = new Feefie(); //default constructor
}
```

# ONLY AVAILABLE WHEN NO OTHERS EXIST

Default constructors are not created if other constructors have been defined.

```
class Baz{ public Baz(int x) {} }
public class Main {
  public static void main(String []args){
    Baz inga = new Baz(); // Error!!!
  }
}
```

# KEYWORD: `THIS`

- Refers to the object that "owns" the current method
- Can't be used in a static context
- Used to call overloaded constructors (from other constructors)

```
public Foo(int x){
    this.x = x;
    this();
}
```

# MEMBER INITIALIZATION

All class members are initialized to default values
0, `false`, or `' '` for primitives.
`null` for object references.
Static members are initialized the first time that class is referenced.
Local variables (inside methods) are not auto-initialized.

# CONSTRUCTOR INITIALIZATION

All instance and static fields initialized before constructor runs.
Use constructors to set custom initial values.

# ARRAY INITIALIZATION

Arrays are objects and will receive a default value of `null`
Two ways to initialize:

```
int[] arr1 = {1, 2, 3} // Array Literal
```

OR

```
int[] arr2 = new int[3]
arr2[0] = 3;
arr2[1] = 2;
arr2[2] = 1;
```

# ARRAY ACCESS

Access array elements with `[index]` where index is a number. Array indexes start at 0. Arrays have a `length` field built in. Arrays are references.

```
int[] arr1 = {1, 2, 3};
int[] arr2 = arr1;
arr2[0] = 5;
System.out.println(arr1[0]);
// output: 5
```

# ENUMERATED TYPES

Enumerated types are a way to create a list of possible values. Created using the `enum` keyword.

```
public enum CompassDirection {NORTH, SOUTH, EAST, WEST }
...
CompassDirection heading = CompassDirection.SOUTH;
```

# CLEANUP

# DESTRUCTORS?

Some languages have a method type called a destructor.
Destructors free up memory when it is no longer needed.
Explicit memory management is complex and prone to bugs.

# NO JAVA DESTRUCTORS

Java uses a Garbage Collector.
Out of scope objects are garbage collected and their memory freed automatically.
Garbage Collector is automatic and thus not predictable.