Java Basics cheatsheet

Key Terms

- Most Java programming involves calling **methods** of **objects**. The syntax for this is `obj.method(param1, param2)`.
- Objects are created when we **instantiate classes**. We do so with the *new* keyword: `new Cat("Ziggy")`. This calls the class **constructor**.
- Classes can **extend** other classes, which means that they inherit all the functionality from those other classes (but they can also overwrite some of it).
- We also have **interfaces** which are a set of method signatures. A class can **implement** an interface if it has implementations for all the methods indicated in the interface.
- The `this` keyword is used in an object method to refer to the object itself, and to provide access to its **fields**.

Visibility

Classes, Variables, and Methods etc have *visibility* indicated by a keyword:

public Can be accessed by anyone.

private Can only be accessed from objects of the class that contains them.

package-private Can be accessed by any classes that are within the same package. This is also the default, if no specific word is used.

protected Can be accessed by subclasses of the class.

Variables

Methods and objects work with a number of different "variable" symbols. They vary in their **scope**, i.e. the specification of all the parts of the code where they exist.

Instance variables also called **fields**, are unique to each object, typically created when the object is instantiated. Their values are shared amongst all the methods of the object.

Static variables or **static fields** are properties associated with a class and are shared amongst all object instances of that class. Similarly static methods can be called just using the class name and without requiring a class instance.

Parameters or **arguments** are passed to the method from its caller. Their value only extends to the end of the specific function call.

Local variables or simply **variables** are defined within a function and exist only within the innermost set of curly braces that contains their declaration.

Type of a variable

- Each variable has a *type* specified at the moment of its declaration.
- This can be a built-in datatype like int or char, or it can be a class or interface.
- When assigning a value to a variable, the type of

Syntax elements

Java files

Java files, with extension .java, consist of two parts:

- a *package* statement indicating which package the file belongs to.
- *import* statements that load public elements from other packages.
- a class or interface definition.

Import statements are used to avoid having to refer to packages by their fully qualified names. Here are some examples, considering an imaginary package graphics that contains a Rectangle class:

```
import graphics.Rectangle;
import graphics.*;
import graphics.Rectangle.*;
```

Without an import, we can use a class by referring to it with its **qualified name**, for example graphics.Rectangle.

The first import line above allows us to refer to the Rectangle class directly in our code from now on. The second import line does the same for *all* files within the graphics package.

The third import line will make every inner class and method within the Rectangle class available directly. For example, if Rectangle had a static make method, then with the first two imports we can refer to it as Rectangle.make(...), while with the last import we can do make(...) instead.

Class definitions

Class Definitions have the keyword class, possibly preceded by a *visibility modifier*, followed by the class name (capitalized). It may be followed by extends ... and/or implements ... clauses, if the class is a subclass of another class or if it implements a certain interface.

```
public class Circle extends AbstractShape implements drawable. {
    ...
}
```

The interior of a class definition may contain any of the following, in any order:

- field declarations
- zero or more constructor definitions
- method definitions
- inner class definitions

Interface definitions are similar, except that they cannot contain an extends part, cannot have constructors, and instead of method definitions they have **method declarations**, containing a semicolon instead of a body.

Abstract class definitions are similar, except that they cannot have constructors, they contain the abstract modifier to their definition, and they can contain both method definitions and method declarations. Methods that are simply declared must contain the abstract modifier.

Field declarations

A field declaration specifies the visibility and type of the field, and possibly an initial value for the field. Some examples

```
private String firstName;
static final int MAX_CAPACITY = 40;
public static String hostname;
```

The first example specifies a private field of type String and called firstName. This field will be likely initialized in the constructor.

The second example specifies what is effectively a *constant*: A static variable, visible within the package (hence no modifier in front), of type int, and set to be **final**, meaning its value cannot change.

The third example is a public static field, so visible everywhere and with a value independent of the specific instance.

Method definitions

Method definitions consist of a number of *modifiers* followed by the method's *return type*, the method name, and a parenthesized parameter list, with their types specified. The body of the method follows inside curly braces.

For example:

```
public void setFirstName(String newFirstName) {
    ...
}

public String getFirstName() {
    ...
}

private static boolean isValidName(firstName, lastName) {
    ...
}
```

An important example is the **main method** that your program can have, which has a specific required signature:

```
public static void main(String[] args) {
    System.out.println("Hello world!");
}
```

Constructors

Constructors are similar to method definitions. They combine the return type and name onto one thing, namely the class name. They cannot be static, and they don't return any values in their body (the newly created object is what is being returned).

A class can have multiple constructors, provided they take different kinds of arguments. It is customary in this case for all constructors to eventually call the same constructor, the one with the most complete set of arguments.

By default, a class with no specific constructors has a *default constructor* `public void ClassName() {}`.

Example:

```
public Person(String firstName, String lastName, int age) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.age = age;
}

// This constructor simply calls the longer constructor.
public Person(String firstName, String lastName) {
    this(firstName, lastName, 0);
}
```

Constructors are called using the `new` keyword: `Person p = new Person("Peter", "Doe", 26);`

Control Flow

Java contains many of the standard control flow elements you may have seen in other languages:

- conditionals
- while loops
- for loops
- switch statements

Conditionals Java conditionals follow a standard `if-then-else` pattern:

```
if (thisIsTrue) {
    ...  true case code
}

if (thisIsTrue) {
    ...  true case code
} else {
    ... false case code
}

if (thisIsTrue) {
    ...  true case code
} else if (thatIsTrue) {
    ... case where thatIsTrue is true but thisIsTrue is false
} else {
    ... both false
}
```

There is also the **ternary operator**, which *returns a value*:

```
int x = (y > 5) ? 5 : y;
```

In this example, x will have the same value as y unless that is more than 5, in which case it is scaled down to a 5.

While loops Java implements the familiar while loops, that execute a block of code as long as a specific condition is true:

```
while (account.hasMoney()) {
    account.withdraw();
}
```

This loop will keep executing the account.withdraw() method as long as account.hasMoney() is true.

For loops Java provides two kinds of for loops. The standard *counter loop* and a *foreach loop* for iterating over a collection.

```
for (int i = 0; i < names.length; i++) {
    System.out.println(names[i]);
}

for (String name : names) {
    System.out.println(names);
}
```

If you are simply looping over the elements of a collection, you should always prefer the second way.

Switch statements A switch statement compares a value against a series of constant values presented in case expressions, and executes the corresponding statements. It will fall through to the next case unless you remember to use break.

```
switch (person.getType()) {
    case "faculty":
        doSomething();
        break;
    case "student":
        doSomethingElse();
        break;
    default:
        // We don't do anything otherwise
}
```