# Overview

Core Classes

Working with class `String`.

Pitfalls when using operator ==

Using `equals(...)`.

# Openjdk source code repository

- Welcome to the JDK!

Java Visualizer

# Java Visualizer

*(beta:* report a bug)

Write your Java code here:

```
1 public class ClassNameHere {
2     public static void main(String[] args) {
3
4     }
5 }
```

http://prog.mi.hdm-stuttgart.de/java_visualize

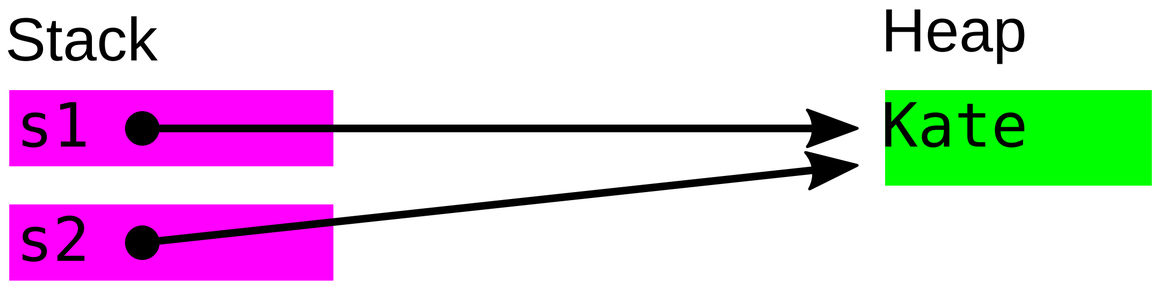
# Superclass `Object`

- Superclass of all Java™ classes.
- Common methods to be redefined by derived classes.

# String literals

```
String s1 = "Kate";

String s2 = "Kate";
```



# OpenJDK String implementation

## Implementation of `java.lang.String`:

```
public final class String ... {
  private final char value[];
  private int hash;
  private static final long serialVersionUID = -6849794470754667710L;
...
}
```

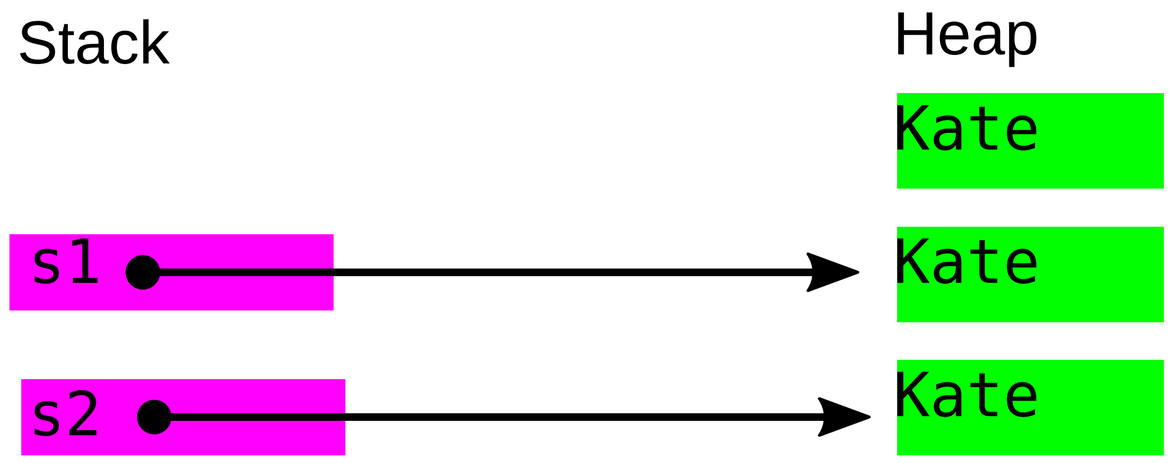
# String copy constructor

| Code | Output |
| --- | --- |
| `final String s = "Eve"; (1)`<br>`final String sCopy = new String(s); (2)`<br>`System.out.println("sCopy == s: " + (sCopy == s)); (3)`<br>`System.out.println("sCopy.equals(s): " + sCopy.equals(s)); (4)` | `sCopy == s: false (3)`<br>`sCopy.equals(s): true (4)` |

# Copy constructor and heap

```
String s1 =
 new String("Kate");

String s2 =
 new String("Kate");
```



# Operator == and `equals()`

| Primitive type | Object |
|---|---|

```java
// equal values
int a = 12, b = 12;

System.out.println(
    "==: " + (a == b));
// No equals(...) methoc
// for primitive types
```

```java
String
 s1 = new String("Kate"),
 s2 = new String("Kate");

System.out.println(
  "    ==: " + (s1 == s2));
 System.out.println(
  "equals: " + s1.equals(s2));
```

---

```
==: true
```

```
    ==: false
equals: true
```

# Remarks `==` vs. `equals()`

- The == operator acting on primitive types compares expression values.
- The == operator acting on objects compares for equality of reference values and thus for object identity.
- The == operator acting on objects does **not** check whether two objects carry semantically equal values.
- The `equals()` method defines the equality two objects.

# Operator == and `equals()` implications

- Each object is equal by value to itself:

  `object1 == object2` ⇒ `object1.equals(object2)`
- The converse is not true. Two different objects may be of common value:

## Code

```
String s = "Hello", copy = new String(s);

System.out.println("equals: " + s.equals(copy
System.out.println("    ==: " + (s == copy));
```

## Result

```
equals: true
    ==: false
```

`equals()` is being defined within respective class!

Implementation at https://github.com/openjdk/ .../String.java :

```
public final class String ... {
public boolean equals(Object anObject) {
  if (this == anObject) {
    return true;
  }
  return (anObject instanceof String aString)
    && (!COMPACT_STRINGS || this.coder == aString.coder)
    && StringLatin1.equals(value, aString.value);
}
```

# Overview

Core Classes

- ➠ Objects, equals() and hash-values

Why using hash values?

`hashCode()` and `equals(...)`.

«Good» `hashCode()` implementations.

# Hashing principle


“I want the 12p one”

Quickly identify by “simple” value

- Where is the blond haired guy?
- I take the pink flower.
- The 334.50$ cellular phone.

# Hashing in Java and `equals()`

Method `hashCode()`: Instance 0 ⇨ o.`hashCode()`, of type `int`.

- Same value on repeated invocation
- Objects with identical value with respect to `equals()` must have identical hash values:
  `true == a.equals(b)` $\Longrightarrow$ `a.hashCode() == b.hashCode()`.
- Conversely: Two instances differing with respect to `equals()` may have identical hash values.

Consequence: `equals()` and `hashCode()` must be **redefined simultaneously**!

# `Rectangle` `equals(...)` and `hashCode()`

```
public class Rectangle {
    int width, height;
    @Override public boolean equals(Object o) {
        if (o instanceof  Rectangle r) {
            return width == r.width
                    && height == r.height;
        } else {
            return false;
        }
    }
    @Override public int hashCode() {
        return width + height;
    }
}
```

# `Rectangle` hash values

```
public class Rectangle {
    int width, height;
...
    @Override public int hashCode() {
        return width + height;
    }
}
```

| width | height | hash value |
|---|---|---|
| 1 | 3 | 4 |
| 2 | 2 | 4 |
| 5 | 5 | 10 |
| 2 | 7 | 9 |
| 4 | 9 | 13 |

# Better `hashCode()` method

```
public class Rectangle {
    int width, height;
...
    @Override public int hashCode() {
        return width + 13 * height;
    }
}
```

| width | height | hash value |
|---|---|---|
| 1 | 3 | 40 |
| 2 | 2 | 28 |
| 5 | 5 | 70 |
| 2 | 7 | 93 |
| 4 | 9 | 121 |

Followup exercises

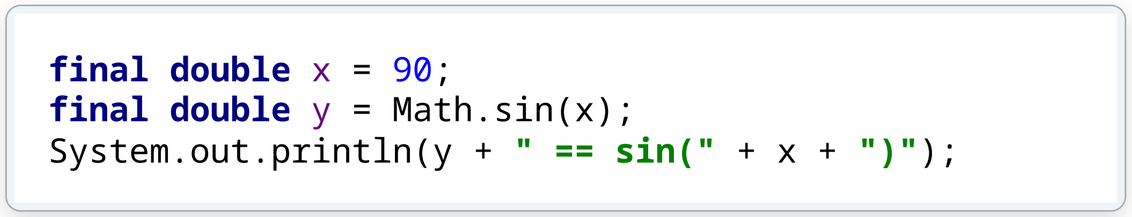
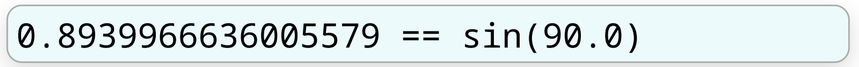
134. Choosing a “good” `hashCode()` method
135. `String` and good hashCode() implementations.

Overview

# Core Classes

➟ Using class `Math`

```
Math.sin(double x)
```

| Code | Result | Math notation |
| --- | --- | --- |
|  |  | $y = \sin x$ |

# Followup exercises

136. Common pitfall using trigonometric functions
137. Using constants from `java.lang.Math`.
138. Strings on CodingBat
139. Masking strings
140. Analyzing strings
141. Pitfalls using “==”: Equality of `String` instances
142. Weird, weirder, weirdest!
143. Analyzing file pathnames