CS133: Developing Programming Principles

Lecture 8

All about references: the memory model, passing parameters, the null reference

`Rectangle` Class

A **`Rectangle`** has a height and a width, and the following methods:

- **`public Rectangle(double h, double w)`**
- **`public void setWidth(double newWidth)`**
- **`public void setHeight(double newHeight)`**
- **`public double getWidth()`**
- **`public double getHeight()`**
- **`public double calcArea()`**
- **`public double calcPerimeter()`**
- **`public boolean equals(Rectangle other)`**
- **`public String toString()`**

`Rectangle`: `equals()` method

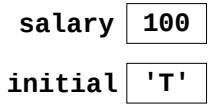
```
// pre: other != null
// post: returns true if this and
// other represent identical
// rectangles
public boolean equals(Rectangle
                      other)
{ return
    (this.height == other.height) &&
    (this.width == other.width);
}
```


How primitive types are stored

- Java code:

```
int salary = 100;
char initial = 'T';
```

- Memory:

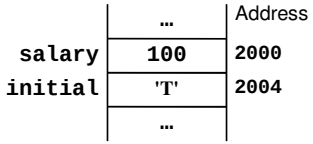


The CS241 View

- Java code:

```
int salary = 100;
char initial = 'T';
```

- Memory:



References

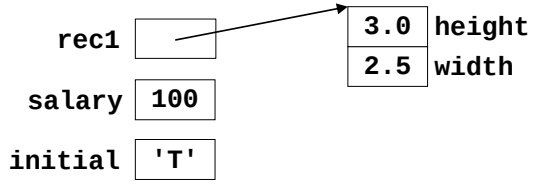
- A reference is the memory address where an object is stored.
- Recall that we typically do not compare objects using == or assign values to them using =.
- Why?

Picture

- Java code:

```
int salary = 100;
char initial = 'T';
Rectangle rec1 = new Rectangle (3.0, 2.5);
```

- Memory:



Picture explained

- Notice the variable **`rec1`** doesn't have the **value** of the **`Rectangle`** in its memory location.
- It has a **reference** to where the value is stored.
- **Crucial concept**

References continued

- When it comes to objects, those two operators (= and ==) compare and assign **references**, not values.
- What does that mean?

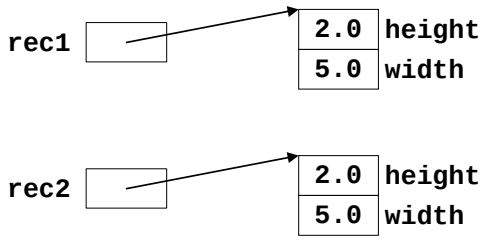
Comparing objects using **==**

- Example:

```
Rectangle rec1 = new Rectangle(2.0, 5.0);
Rectangle rec2 = new Rectangle(2.0, 5.0);
1 System.out.println(rec1 == rec2);
2 System.out.println(rec1.equals(rec2));
```

Why did we get this output?

Picture



Picture explained

- So when we compare:

  `rec1 == rec2`

  We are asking whether the references point to the same object.

- When creating an object, we want an equals **method which looks at the object's variables**.

Assigning references

```
Rectangle orig = new Rectangle(7.0, 6.25);
Rectangle sameSize = new Rectangle(7.0, 6.25);
Rectangle different = new Rectangle(3.0, 5.6);

System.out.println(orig == sameSize);
// will print "false" because the references
// point to different objects in memory

different = orig;
System.out.println(different == orig);
// will print "true" because both references
// point to the same object
```


Passing parameters

- In Java, parameters are **passed-by-value**.
  - That is, we copy the argument to the parameter variable.
- Primitives: copy the variable
- Objects: copy the *reference*
- *Implication: we cannot modify the argument variable/reference*
  - *... but we **can** modify the objects pointed to by references!*

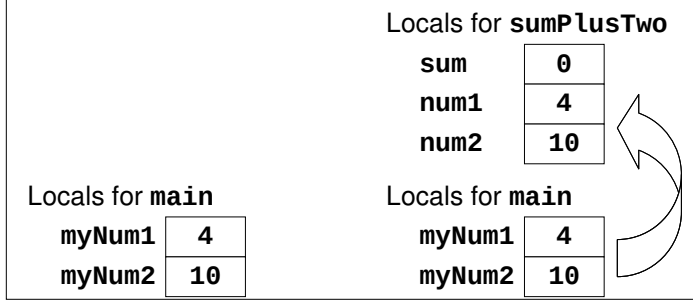
Passing parameters: primitive types

```
public static int sumPlusTwo(int num1, int num2)
{
  int sum = 0;
  num1 = num1 + 1;
  num2 = num2 + 1;
  sum = num1 + num2;
  return sum;
}

public static void main(String[] args) {
  int myNum1=4, myNum2 = 10;
  System.out.println(sumPlusTwo(myNum1,myNum2));
  System.out.println(myNum1);
  System.out.println(myNum2);
}
```


Explanation

- When actual parameters are passed, a **copy** of the **value** is given to the formal parameter.



Explanation continued

- The called method changes the formal parameters (the copies), not the originals.

Locals for **`sumPlusTwo`**

`sum`	**`16`**
`num1`	**`5`**
`num2`	**`11`**

Locals for **`main`**

`myNum1`	**`4`**
`myNum2`	**`10`**

Explanation continued

- When the method finishes, its local variables and parameters disappear. Changes are lost.

Locals for **`main`**

`myNum1`	**`4`**
`myNum2`	**`10`**

Object parameters

- When passing objects as parameters, the reference is passed.
- Any changes to the object within the method will remain once it finishes executing.

`Rectangle` manipulation

```
public class RectangleChanger {
  public static void doubleWidth(Rectangle r) {
    r.setWidth(2 * r.getWidth());
  }
  public static Rectangle halveHeight(Rectangle r)
  {
    return new Rectangle(
         r.getHeight()/2.0, r.getWidth());
  }
  public static void makeSquare(Rectangle r) {
    r = new Rectangle(r.getWidth(), r.getWidth());
  }
```

`Rectangle` manipulation continued

```
  public static void main(String[] args) {
    Rectangle troy = new Rectangle(2.0, 4.0);
    doubleWidth(troy);
    Rectangle shortTroy = halveHeight(troy);
    makeSquare(troy);
  }
} // end class
```


`Student` class example

```
public class Student {
  private int studentID;
  private String name;
  public void setName(String newName) {
    this.name = newName;
  }
  public String getName() {
    return this.name;
  }
  public int getStudentID {…}
  public void setStudentID {…}
}
```

Example continued

```
public class TestStudent {
  public static void changeName(Student s) {
    s.setName("Perry Hotter");
  }
  public static void main(String[] args) {
    Student troy = new Student();
    troy.setName ("Troy Vasiga");
    System.out.println("Before:"+troy.getName());
    changeName(troy);
    System.out.println("After:"+troy.getName());
} }
```


`null`

- **`null`**: a special holder "value" that can be used for any class type (object).
- Actually is "no value".
- Use **==** and **!=** to compare objects to null.

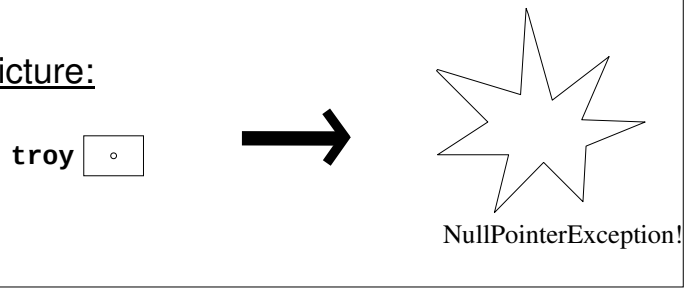
`NullPointerException`

- When trying to do anything (like call a method) with an object that **`== null`**, the result is this error.
- Means the object is not initialized, and thus can't do anything.

Null pointer example

```
Student troy = null;
troy.setName("Troy Vasiga");
```

Picture:



Summary

- References
- Object parameters
- Methods calling methods
- **`null`**
- **`NullPointerException`**s

ERROR: undefined
OFFENDING COMMAND:

STACK: